Cryptography and Cryptanalysis in Java

Creating and Programming Advanced Algorithms with Java SE 21 LTS and Jakarta EE 11

—

Second Edition

—

Stefania Loredana Nita
Marius Iulian Mihailescu

Apress®

Cryptography and Cryptanalysis in Java

Creating and Programming Advanced Algorithms with Java SE 21 LTS and Jakarta EE 11

Second Edition

Stefania Loredana Nita
Marius Iulian Mihailescu

Apress®

Cryptography and Cryptanalysis in Java: Creating and Programming Advanced Algorithms with Java SE 21 LTS and Jakarta EE 11

Stefania Loredana Nita
Bucharest, Romania

Marius Iulian Mihailescu
Bucharest, Romania

ISBN-13 (pbk): 979-8-8688-0440-3
ISBN-13 (electronic): 979-8-8688-0441-0
https://doi.org/10.1007/979-8-8688-0441-0

Copyright © 2024 by Stefania Loredana Nita, Marius Iulian Mihailescu

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Melissa Duffy
Development Editor: Laura Berendson
Editorial Project Manager: Gryffin Winkler

Cover designed by eStudioCalamar

Cover image designed by Towfiqu barbhuiya on Unsplash

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

Table of Contents

About the Authors ix

About the Technical Reviewer xi

Chapter 1: Getting Started 1

Theoretical Cryptography vs. Applied Cryptography 1

Theoretical Cryptography 2

Applied Cryptography 3

Cryptography and Cryptanalysis 8

Book Structure 9

Conclusion 17

References 17

Chapter 2: JDK 21: New Features 19

Conclusion 35

References 36

Chapter 3: Roadmap and Vision for Jakarta EE 11 39

Conclusion 53

References 54

Chapter 4: Java Cryptography Architecture 55

Architecture and Design Principles 58

JCA Classes and Algorithms 63

Algorithms and Engine Classes 63

Interfaces and Main Classes 64

Data Encryption 65

Hash Functions 67

Signatures 69
Conclusion 75
References 75

Chapter 5: Classical Cryptography 77

Caesar Cipher 78
Implementation 79
Cryptanalysis 83
Vigenère Cipher 83
Implementation 86
Cryptanalysis 89
Hill Cipher 89
Implementation 91
Cryptanalysis 96
Conclusion 96
References 97

Chapter 6: Quantum Cryptography 99

Key Concepts for Quantum Mechanics Used in Cryptography 99
Quantum Superposition 99
Quantum Entanglement 102
Heisenberg's Uncertainty Principle 105
Quantum Key Distribution 108
Security and Eavesdropping Detection 108
Challenges 109
BB84 Protocol 109
Security Based on Quantum Principles 110
Description of BB84 Protocol Simulation 111
Other Quantum Cryptography Algorithms 113
Conclusions 115
References 115

Chapter 7: Formal Techniques for Cryptography 117
Definitions 117
Probabilities and Statistics 119
Conditional Probability 120
Random Variables 120
Entropy 121
A Little Algebra 122
Elliptic Curves 128
Test Suites 130
Conclusion 131
References 132

Chapter 8: Pseudorandom Number Generators 135
Examples of PRNGs 138
Linear Congruential PRNGs 138
Blum-Blum-Shub PRNG 139
Linear Circuit PRNGs 140
Other PRNGs 141
PRNGs Security 142
Java.util.Random Class 142
Example of Using CSPRNGs 146
Conclusion 147
References 148

Chapter 9: Hash Functions 151
Conclusion 160
References 161

Chapter 10: Symmetric Encryption Algorithms 165
Data Encryption Standard 166
The Generation of Keys 168
Encryption and Decryption Process 169
Operation Modes for DES 170

Advanced Encryption Standard 175
Encryption and Decryption Process 176
Operation Modes for AES 176
Conclusion 181
References 182

Chapter 11: Asymmetric Encryption Schemes 183

RSA 185
ElGamal 191
Merkle-Hellman 197
The Knapsack Approach 197
Algorithms 197
Advancements in Asymmetric Cryptography 199
Elliptic Curve Cryptography 199
Post-Quantum Cryptography 200
Conclusion 201
References 201

Chapter 12: Advanced Encryption Schemes 205

Homomorphic Encryption 205
Searchable Encryption 210
Conclusion 212
References 213

Chapter 13: Identification Schemes 219

FFS Identification Protocol 223
Conclusion 231
References 232

Chapter 14: Signature Schemes 233

Implementations of Digital Signatures 235
Standards and Regulations of Digital Signatures 242
Conclusion 243
References 244

Chapter 15: Lattice-Based Cryptography and NTRU 247

Practical Implementation of the NTRU Library 250

Secure Application Development 266

Conclusion 267

References 267

Chapter 16: Cryptography Tools 269

CryptTool 269

OpenSSL 280

Integrating OpenSSL with Java 286

Conclusion 289

References 289

Index 291

About the Authors

Stefania Loredana Nita PhD, is a lecturer at "Ferdinand I" Military Technical Academy of Bucharest and a software developer and researcher at the Institute for Computers. Her PhD thesis was on advanced cryptographic schemes using searchable encryption and homomorphic encryption. At the Military Technical Academy she teaches courses on formal languages and translators and database application development. She was also an assistant lecturer at the University of Bucharest where she taught courses on subjects such as advanced programming techniques, simulation methods, and operating systems. Her research activity is in the cryptography field, with a focus on searchable encryption and homomorphic encryption. She is also interested in blockchain, quantum cryptography, machine learning, and artificial intelligence. At the Institute for Computers she is working on research and development projects that involve cloud computing security, the Internet of Things, and big data. She has authored and co-authored more than 28 papers at conferences and in journals and has co-authored five books. She holds an MSc degree in software engineering and two BSc degrees, with one in computer science and one in mathematics.

Marius Iulian Mihailescu PhD, has worked in academic and industry sectors for more than 15 years. Currently, he is an associate professor (senior lecturer) in the Faculty of Engineering and Computer Science at "Spiru Haret" University, Romania, and also has a side job as project manager at the Institute for Computers where he is managing projects using different technologies, such as DevOps, Scrum, Agile, C#, Microsoft SQL Server, Syncfusion, ASP.NET, and VUE. At the university, he taught several key computer science courses, about topics such as information security, functional programming, Internet of Things, blockchain, software development methods (Microsoft

Azure, Entity Framework, NHibernate, Linq-to-SQL, UX with DevExpress controls, etc.), and development web applications (HTML 5, CSS 3, Bootstrap, JavaScript, AJAX, nodeJS, VUE, Laravel, mRabbit, ASP.NET, PHP). He authored and co-authored more than 30 articles at conference proceedings, 25 articles in journals, and six books. For three years he worked as IT officer at Royal Caribbean Cruises Ltd, where he dealt with IT infrastructure, data security, and satellite communication systems. He received his PhD in 2014 with a thesis on applied cryptography over biometrics data. He holds two MSc degrees in information security and software engineering, from "Ferdinand I" Military Technical Academy and University of Bucharest, Romania.

About the Technical Reviewer

Valentina Marascu is currently a lecturer in the Faculty of Engineering and Computer Science at "Spiru Haret" University, Romania, and a scientific researcher 3rd degree in the Low Temperature Plasma Laboratory at the National Institute for Laser, Plasma, and Radiation Physics (INFLPR) in Magurele, Romania. Her educational background involves mathematics, physics, and computer science, with proactive engagement in scientific research and the academic sector. Concerning the research sector, she is engaged in scientific experimental physics research, in the frame of fusion and quantum national and international project directions. She has significant scientific papers, which were published in important ISI journals. Concerning the academic sector, her disciplines are web applications development, functional programming, and computational methods in natural sciences, and she is involved in researching and creating techniques to improve cybersecurity measures, with a specific emphasis on analyzing user behavior to detect and minimize potential risks. Complementary to her academic and scientific research activities, her occupations include teacher for experimental physics laboratories, internships, and training with undergraduates and graduate students; chair at various scientific events; jury member at national and international conferences; member of the organizational committee of international conferences; reviewer for different journals; and guest editor, volunteer, and exhibitor participant at scientific events.

CHAPTER 1

Getting Started

In the last decade, technology has rapidly evolved. Statistics show that 64.2 zettabytes of data were generated in 2020 (1 zettabyte is equivalent to 10^{21} bytes or 10^{12} gigabytes), and it is predicted that by 2025, the digital data generated will reach 181 zettabytes [1, 2]. Electronic communication has become an essential part of our lives, and due to its rapid evolution, all manner of security issues has arisen. Because digital messages, in all of their forms, are sent daily over public networks across the world, the need for secure channels and security mechanisms has also increased. Digital devices and communications should have digital signatures that make them easy to authenticate. Modern cryptography provides solutions for all these requirements.

The era in which we are living is considered the "zettabytes era," in which technology allows humans and electronic devices to generate and send information instantly, at any time and any place. Advanced technologies, such as the Internet of Things, fog computing, edge computing, smart vehicles, drones, smart houses, and many other complex software (desktop/web/mobile) solutions or architectures, are evolving so quickly that it is difficult to keep up with security requirements. For example, at the time of this book's writing, there are 160,974 records of vulnerabilities registered on the CVE platform [3]. However, lessons can be learned even from failures, so by analyzing such vulnerabilities, security solutions can be improved.

Theoretical Cryptography vs. Applied Cryptography

Applied cryptography and theoretical cryptography are essential components of the greater science of cryptography, each playing unique but interrelated functions. Comprehending the distinctions between them helps provide an understanding of how cryptographic systems are created, examined, and put into practice.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_1

Theoretical Cryptography

Theoretical cryptography is a branch of cryptography that focuses on the fundamental principles, mathematical theories, and abstract models underlying cryptographic algorithms and protocols. It is concerned with the rigorous analysis and design of cryptographic systems, aiming to establish their security based on well-defined assumptions and formal proofs. The following are key aspects of theoretical cryptography:

- **Mathematical foundations.** Theoretical cryptography heavily relies on advanced mathematics, including number theory, algebra, probability theory, and complexity theory. These mathematical tools are used to create and analyze cryptographic algorithms.
- **Security proofs.** A crucial part of theoretical cryptography is the development of proofs that demonstrate the security of cryptographic systems under certain assumptions. These proofs are meant to be rigorous and provide a high level of confidence in the security of the systems.
- **Algorithm design.** The design of cryptographic algorithms in theoretical cryptography is not just about creating methods for encryption and decryption but also about ensuring these methods can withstand various types of theoretical attacks. This involves designing algorithms that are computationally hard to break without the proper keys or information.
- **Cryptographic protocols.** Beyond individual algorithms, theoretical cryptography also deals with the creation and analysis of protocols for secure communication and data handling. This includes key exchange protocols, authentication protocols, and more.
- **Computational hardness assumptions.** The security of cryptographic systems in theoretical cryptography often relies on computational hardness assumptions. These are assumptions about the difficulty of solving certain mathematical problems (like factoring large numbers or computing discrete logarithms) within a reasonable amount of time.

- **Abstract models.** Theoretical cryptography uses abstract models to represent and analyze cryptographic systems. These models simplify the real world into a set of assumptions and rules that make analysis manageable while still being relevant to real-world scenarios.
- **Innovative research.** Theoretical cryptography is at the forefront of developing new cryptographic concepts and techniques. It explores the boundaries of what is possible in secure communications and data protection, often paving the way for future practical applications.

Theoretical cryptography involves the creation of novel cryptographic ideas and methods. The focus is on progressing the field rather than on quick practical implementation. Theoretical cryptography establishes the foundation for comprehending cryptographic security principles, whereas applied cryptography implements these principles to develop secure and operational systems in practical scenarios. Both are essential for the progress and real-world implementation of cryptographic systems.

Applied Cryptography

Applied cryptography is a field that deals with the practical implementation and utilization of cryptographic methods in real-world scenarios. It entails utilizing the ideas and theories of theoretical cryptography to design secure communication systems, data protection mechanisms, and other security solutions. Essential components of applied cryptography consist of the following:

- **Implementing cryptographic algorithms.** This process entails converting cryptographic algorithms from theoretical concepts into functional code. Applied cryptography focuses on effectively and securely implementing these methods in software and hardware.
- **System integration.** Applied cryptography focuses on the integration of cryptographic techniques into complex systems. This involves comprehending the integration of encryption inside network protocols, software applications, and hardware devices.

Applied cryptography must address real-world limits such as processing power, memory constraints, and usability issues, unlike theoretical encryption that functions in an idealized setting.

Applied cryptography focuses on assuring the security of cryptographic systems against real assaults, rather than only theoretical ones. This involves factors such as side-channel assaults, in which attackers can extract information from the physical execution of a cryptographic system.

Applied cryptography frequently necessitates conformity to industry norms and governmental mandates about standardization and compliance. This guarantees compatibility among systems and adherence to legislative mandates on data protection and privacy.

Applied cryptography designs and enforces protocols customized for particular uses, like secure web browsing (TLS/SSL), secure email (PGP, S/MIME), and secure financial transactions.

- **User interface and experience.** It encompasses creating systems that prioritize both security and user-friendliness. Users should be able to utilize cryptographic systems proficiently without needing extensive understanding of the underlying cryptography.
- **Update and maintenance.** Applied cryptography is the continuous process of upgrading and maintaining cryptographic systems. With the discovery of new weaknesses and the advancement of computational power, it is essential to upgrade cryptographic algorithms and implementations to ensure security.

Applied cryptography involves implementing the theoretical principles of cryptography in practical scenarios. It connects the theoretical aspects of cryptography with the actual requirements of individuals and organizations to protect their data and communications.

Applied cryptography is essential in the contemporary digital realm for various reasons. They can be summarized as follows:

- **Data security.** Cryptography is crucial for ensuring data security, particularly for protecting sensitive information such as bank records, personal details, and confidential communications. Encryption safeguards data by restricting access to authorized users, preventing unauthorized access or theft.

- **Privacy.** Privacy is crucial in a time where personal data is consistently gathered and analyzed, with encryption playing a vital role in safeguarding it. It enables secure communication and storage of information, free from interception or misuse.
- **E-commerce and online transactions.** Cryptography ensures secure online transactions, which are essential for e-commerce. It guarantees the secure transmission of financial information such as credit card data and authenticates transactions, fostering trust in online commerce.
- **Authentication and identity verification.** Cryptography is utilized for authentication and identity verification to confirm the identification of users and devices, allowing only authorized individuals to access certain services or information. It is essential to avoid identity theft and unwanted access.
- **Integrity of information.** Cryptographic techniques are utilized to maintain the integrity of information by preventing any unauthorized alterations or tampering during transmission or storage. It is crucial for upholding trust in digital communications and records.
- **National security and defense.** Cryptography is essential for securing government and defense communications for national security and defense purposes. Ensuring secure communication channels is crucial for national security to safeguard sensitive information from possible attackers.
- **Internet and network security.** Cryptography is fundamental to Internet and network security. It secures data during transmission across the Internet and other networks, defending against cyber risks such as hacking and eavesdropping.
- **Digital signatures and nonrepudiation.** Digital signatures use cryptography to verify the source of a digital message or document, preventing the sender from denying sending it (nonrepudiation). This is crucial in legal and corporate settings.

- **Practical implementation.** Applied cryptography involves implementing cryptographic algorithms and protocols in real-world systems. It entails utilizing theoretical principles to create practical cryptography solutions. Integration with systems involves incorporating cryptographic approaches into software and hardware systems, taking into account factors such as performance, usability, and compatibility.

Applied cryptography considers practical constraints and difficulties in creating cryptographic systems, including processing resources, user interface design, and resistance to side-channel attacks. Ensuring that cryptographic implementations adhere to standards and laws is vital for interoperability and legal compliance.

Practical security involves implementing and operating secure systems in real-world situations, considering unexpected weaknesses and practical limitations, in contrast to the theoretical foundation provided by cryptography.

One of the most important aspects considered when complex systems are designed and implemented is knowledge. In antiquity, the Latin saying *Scientia potentia est* means "Knowledge is power" [4]. In the 21st century, this is even more true; information falling into the wrong hands can lead to huge business losses and catastrophic outcomes. Cryptography and information security provide security mechanisms that can protect information shared between senders and recipients over insecure channels so that unauthorized users cannot access or alter the transmitted information. Over time, there were encryption systems that were broken by attackers by exploiting vulnerabilities of the systems.

The word *cryptography* comes from the Greek words *kryptos* and *graphein*, meaning "hidden" and "writing," respectively. As its name suggests, the purpose of cryptography is to hide messages from unauthorized individuals and to keep their integrity. Although the study of cryptography has been around only about 100 years, it was used in different forms from ancient times. However, over time there have been various primary methods of hiding secret messages, starting with hieroglyphs, continuing with Caesar's famous cipher, and followed by the Vigenère cipher, Hebern's rotor machine, and the famous Enigma machine. Nevertheless, hiding messages was not the only occupation close to cryptography or, rather, information security. Another example is authentication or identity verification; this was often done through seals.

Cryptography is considered an art, especially in its primary phases. The history of cryptography began in ancient Egypt, alongside the art of writing, during a time when humans started organizing in different social groups. This organization led to a natural need to transmit information only to certain individuals, with the same group, tribe, etc. Early forms of cryptography were hieroglyphs, which started to be used about 4,000 ago by Egyptians—only they recognized the symbols and their meaning. An inscription carved circa 1900 BC contains the first known evidence of cryptography (in some kind). It is located in Egypt nobleman Khnumhotep II's tomb, in the main chamber [5]. In this inscription, some symbols have a different form than usual, and the scribe's intent was not necessary to hide a message; rather, he wanted the symbols to look nobler than usual according to the social status of the deceased. Although the inscription does not hide a message, it contains an altered/transformed form of the original symbols, being the oldest proof of such an approach.

Then, cryptography in the ancient world moved to a substitution approach, in which every symbol of an alphabet was replaced by another symbol based on a secret rule. This happened around 500–600 BC. The next notable cipher was Caesar's cipher. Caesar was a Roman emperor who was communicating with his army generals with encoded messages, using a substitution within the Roman alphabet. Each letter was shifted a certain number of positions in the alphabet, usually three. For example, the correspondent of A was D, of B was E, and so on. This is an important historical cipher that is mentioned often in cryptography literature.

The next important achievement in cryptography was in the Middle Ages by Leon Battista Alberti, who implemented polyalphabetic substitution. Two rotating copper disks were used that had the alphabet inscribed on them. Different variations of polyalphabetic substitution ciphers were used, but the most known such cipher is Vigenère.

Then in the 19th century, the encryption methods evolved and became more technical. The beginning of modern cryptography starts mainly with the Enigma machine, although the rotors were used a few years before Enigma's invention. Considered unbreakable, the Enigma machine was invented by German engineer Arthur Scherbius at the end of World War I, but it was extensively used in World War II by the German army. Enigma is based on more rotors that work electromechanically, and it scrambles the letters of the alphabet. During World War II, cryptography alongside cryptanalysis evolved quickly and became mathematized. Then modern cryptography continued with symmetric encryption (using a private key for both encryption and

decryption) and was followed by asymmetric encryption (where a public key is used for encryption and a private key for decryption) introduced by Diffie and Hellmann in 1976. Since then, different types of encryption systems evolved from each type of cryptography (symmetric or asymmetric).

For a more detailed history or interesting facts about cryptography, you can consult [6] and [7].

The book aims to present the main topics of cryptography, information security, and cryptanalysis from a practical perspective, by providing examples of implementations in Java. The book addresses a large audience, such as security experts, military experts and researchers, ethical hackers, teachers in academia, researchers, software developers, and software engineers, and it can represent a good starting point in developing secure applications, together with [8], [9], and [10].

Cryptography and Cryptanalysis

When working with information security and data protection, the concepts of cryptology, cryptography, and cryptanalysis should be clear. These are defined here, as presented in [8, 9]:

- *Cryptology* is defined as the science or art of secret writings; the main goal is to protect and defend the secrecy and confidentiality of the information with the help of cryptographic algorithms.
- *Cryptography* represents the defensive side of cryptology; the main objective is to create and design cryptographic systems and their rules. When we are dealing with cryptography, we can observe a special kind of art, an art that is based on protecting the information by transforming it into an unreadable format, called *ciphertext*.
- *Cryptanalysis* is the offensive side of cryptology; its main objective is to study the cryptographic systems with the scope to provide the necessary characteristics in such a way to fulfill the function for which they have been designed. Cryptanalysis can analyze the cryptographic systems of third parties through the cryptograms realized with them, breaking them to obtain useful information for their business purpose. Cryptanalysts, code breakers, or ethical hackers are the people who are in the field of cryptanalysis.

- *Cryptographic primitive* represents a well-established or low-level cryptographic algorithm used to build cryptographic protocols. Examples of such routines include hash functions or encryption functions.

Book Structure

This book contains 15 chapters, in which the main aspects of classical and modern cryptography are presented. Generally, the chapters will cover the foundation of the presented concept/mechanism/technique from a mathematical perspective and then a practical implementation or use cases in Java. The chapters are detailed here:

- **Chapter 2, "JDK 21: New Features".** This chapter explores the newest features in JDK 21, a major update in the Java Development Kit series. The chapter provides an introduction to JDK 21, emphasizing the significance of ongoing improvements in the Java ecosystem. The chapter systematically examines every new feature, upgrade, and API update in this version, offering a thorough understanding of their functioning and influence on Java development. The main areas of attention involve enhancements to the language, such as new syntax or language constructs, optimizations in the Java Virtual Machine (JVM) to increase performance and efficiency, and extensions in the standard library to provide more powerful and flexible programming choices. The chapter emphasizes enhancements in security features, demonstrating Java's continuous dedication to secure programming techniques. It also includes the progress in developer tooling and support to enhance the developer experience and productivity. Each feature is illustrated with actual examples and use cases to show how these upgrades can be efficiently applied in real-world Java programs. Java developers, software architects, and technology aficionados should read this chapter to remain updated on the newest trends and capabilities in the Java platform and make the most of JDK 21 in their software development projects.

- **Chapter 3, "Roadmap and Vision for Jakarta EE 11".** This chapter will present the basic usage of Jakarta EE and explain how security mechanisms can be integrated. This chapter provides an in-depth analysis of the plan and vision for Jakarta EE 11, emphasizing the strategic direction and significant enhancements in the latest version of the enterprise Java platform. The text provides a summary of Jakarta EE's development, highlighting its significance in today's corporate software environment. The chapter outlines the goals and objectives for Jakarta EE 11, focusing on enhancing cloud-native Java application development, improving scalability, and increasing developer productivity. The main characteristics and changes in Jakarta EE 11 are comprehensively analyzed, focusing on improvements in APIs, new functionality, and enhanced compatibility with various technologies and platforms. The chapter explores the community-driven strategy in Jakarta EE's development, highlighting the collaborative work among individual contributors, enterprise stakeholders, and the wider Java community. Focus is placed on incorporating new technologies and trends like microservices architectures, containerization, and DevOps approaches to keep Jakarta EE up-to-date and forward-thinking. The chapter examines the problems and opportunities facing Jakarta EE and assesses its standing in the competitive field of enterprise application development. This chapter offers a thorough summary and a future-oriented examination, making it a valuable tool for enterprise developers, software architects, and IT decision-makers as they navigate the changing landscape of Java-based enterprise solutions and strategize their implementation of Jakarta EE 11.
- **Chapter 4, "Java Cryptography Architecture".** This chapter presents the built-in functions of Java that can be used in cryptography. These are encapsulated in Java's cryptography application programming interface (API) called Java Cryptography Architecture (JCA). The chapter offers an in-depth examination of the Java Cryptography Architecture (JCA), a crucial framework in the Java Development Kit (JDK) for implementing cryptographic algorithms. The chapter begins with an introduction to the JCA, outlining its significance in

providing a secure environment for Java applications. It explains the core concepts and components of the JCA, including the provider architecture, cryptographic service classes, and key management. The chapter then delves into the detailed workings of cryptographic algorithms supported by JCA, such as encryption and decryption algorithms, digital signatures, message digests, and secure random number generation. It also covers the use of key factories, key generators, and key agreement protocols within the JCA framework. The practical implementation of these cryptographic techniques in Java applications is thoroughly explored, with examples illustrating how to utilize JCA for common cryptographic tasks. The chapter highlights the importance of understanding security best practices and the nuances of implementing cryptographic solutions effectively in Java. Additionally, it discusses the extensibility of JCA, explaining how developers can integrate third-party cryptographic providers or develop their own. The chapter concludes by addressing the evolving landscape of cryptography, such as the advent of quantum-resistant algorithms, and how the JCA is adapting to these new challenges. This chapter serves as both a technical guide and a strategic resource for Java developers, security professionals, and system architects who seek to enhance the security of their Java applications using the Java Cryptography Architecture.

- **Chapter 5, "Classical Cryptography".** This chapter will describe classic encryption and decryption techniques for certain ciphers. These ciphers use basic mathematical functions but represent a good starting point in understanding the purpose of cryptography and its basic rules. For each classic algorithm, the mathematical description will be presented followed by the implementation in Java. Also, the chapter offers a detailed summary of classical cryptography, which serves as the basis for contemporary cryptographic methods. The text traces the historical development of cryptography, from ancient ciphers to the advanced systems of the early 20th century. The main emphasis is on traditional encryption techniques such substitution ciphers such as the Caesar cipher and the Vigenère cipher, as well as

transposition ciphers, with a focus on their historical evolution and use. The chapter explores the foundations and mechanics of classical cryptographic approaches, detailing their functionality and historical use in securing communication.

- **Chapter 6, "Quantum Cryptography".** This chapter offers detailed guidance on how to create quantum cryptography using Java. The text introduces the basic ideas of quantum computing, such as qubits, superposition, and entanglement, which are crucial for comprehending quantum cryptography. Next, the attention turns to simulating quantum systems, recognizing the restricted access to quantum computers. The chapter examines Java libraries that may simulate quantum settings, emphasizing their importance in creating quantum cryptography applications. The main focus is on quantum key distribution (QKD), which demonstrates how it allows for the creation of secure keys using principles of quantum mechanics. The chapter also covers the incorporation of quantum cryptography with conventional cryptographic techniques. Quantum methods are mostly utilized for key distribution, while classical cryptography is responsible for securing data transmission. The chapter discusses the challenges and limitations of quantum cryptography, including issues related to key generation rates and transmission distance constraints. It stresses the significance of keeping abreast of the fast-changing subject of quantum computing and cryptography. This chapter is a crucial resource for Java programmers and cryptography aficionados interested in exploring the advanced field of quantum cryptography.
- **Chapter 7, "Formal Techniques for Cryptography".** This chapter is focused on the formal aspects of cryptography. It will present, without going into too many technical details, the main mathematical elements used in cryptography that are mandatory to understand; it will also define some specific terms of cryptography. You will find a detailed examination of formal techniques in cryptography, highlighting their crucial importance in creating and validating cryptographic protocols and algorithms. The text introduces formal methods and their importance in cryptographic

systems, emphasizing the significance of security and accuracy. The chapter delves into many formal methods employed in cryptography, such as mathematical modeling, logic-based procedures, and automated theorem proving. These approaches are essential for demonstrating the security attributes of cryptographic algorithms and protocols, including secrecy, integrity, and authenticity.

- **Chapter 8, "Pseudorandom Generators".** Randomness is one of the most important concepts used in cryptography. This chapter will present what pseudorandom generators are and why they are important in cryptography. One section of the chapter is dedicated to the `java.util.Random` class. This chapter explores the concept and use of pseudorandom generators (PRGs) in cryptography and computer science. The text starts by defining pseudorandomness and discussing the importance of pseudorandom generators (PRGs) in different cryptographic scenarios. The chapter explains the theoretical basis of PRGs, highlighting their differences from true random number generators and the mathematical requirements for a PRG to be considered cryptographically secure. The discussion transitions to the creation and assessment of PRGs, exploring the typical algorithms and methods employed to produce pseudorandom sequences that cannot be differentiated from genuinely random sequences within a polynomial timeframe. The topics covered are linear congruential generators, linear feedback shift registers, and advanced cryptographic constructs such as the Blum Blum Shub and Mersenne Twister algorithms. The chapter primarily focuses on showcasing the practical application of the `java.util.Random` class in Java programming. The text contains code samples and situations in which `java.util.Random` is commonly used, as well as recommended methods for its application. The chapter discusses the shortcomings of `java.util.Random`, especially in situations that demand high-security random number generation, including cryptographic key generation or secure token production.

- **Chapter 9, "Hash Functions".** This chapter explores how hash values can be generated for different types of data and presents some important hash functions, like the MD or SHA families. This chapter provides an in-depth analysis of hash functions, a key subject in the realms of cryptography and computer science. The text starts by defining hash functions and elaborating on their crucial importance in applications including data integrity verification, password storage, and digital signatures. The chapter describes the fundamental characteristics of cryptographic hash functions, such as determinism, preimage resistance, second preimage resistance, and collision resistance. The discussion transitions to several hash functions, offering a summary of frequently utilized algorithms like MD5, SHA-1, and the SHA-2 series. The chapter explores the design concepts, cryptographic strength, and historical views of various algorithms, including their evolution and flaws.
- **Chapter 10, "Symmetric Encryption Algorithms".** This chapter offers a comprehensive analysis of symmetric encryption methods, which are essential in the realm of cryptography. The text introduces symmetric encryption, which involves using a single key for both encrypting and decrypting data. The chapter explores the historical background of symmetric encryption, detailing its progression from ancient ciphers to sophisticated contemporary algorithms. You will delve into the fundamental components of symmetric encryption algorithms, such as their design concepts, operational procedures, and the trade-off between security and performance. The chapter provides a detailed examination of many symmetric algorithms, including the Data Encryption Standard (DES), Advanced Encryption Standard (AES), and others such as Blowfish and Twofish. The structure, operating mode, key management, security features, flaws, and historical importance of each algorithm are thoroughly examined.
- **Chapter 11, "Asymmetric Encryption Schemes".** This chapter highlights the differences between symmetric and asymmetric cryptography and describes two of the most important cryptosystems of this category: RSA and ElGamal. Also, the chapter presents the Merkle-Hellman knapsack system, which is interesting due to its approach.

- **Chapter 12, "Advanced Encryption Schemes".** This chapter delves deeply into the Advanced Encryption Standard (AES), a fundamental component of contemporary cryptographic security. The text provides a historical account of the transition from the old Data Encryption Standard (DES) to AES, outlining the difficult selection process and the rationale for adopting AES as a global standard. The chapter elucidates the fundamental principles of AES, emphasizing its block cipher process, which involves encrypting data in fixed-size blocks, and its utilization of symmetric key cryptography. The composition of AES is thoroughly analyzed, including its key sizes (128, 192, and 256 bits), the encryption rounds for each key size, and the fundamental mathematical operations like substitution-permutation networks. The chapter details the algorithm's steps: SubBytes, ShiftRows, MixColumns, and AddRoundKey, illustrating their role in enhancing AES's resistance to cryptographic assaults.
- **Chapter 13, "Identification Schemes".** This chapter offers a thorough examination of identifying schemes, a vital element in the field of digital security and authentication. The text starts by defining identification schemes and their importance in confirming the identity of individuals or organizations in different digital interactions. The chapter explains the basic principles of authentication and identity, focusing on notions such as uniqueness, reliability, and scalability. The text provides an in-depth analysis of several identification techniques, including knowledge-based systems (e.g., passwords and PINs), token-based systems (e.g., smart cards and key fobs), and biometric systems (e.g., fingerprints, facial recognition, etc.). The chapter explores the technological, procedural, and algorithmic components of these systems, including their implementation, strengths, and flaws.
- **Chapter 14, "Signature Schemes".** This chapter thoroughly examines signature schemes, a crucial component of cryptographic systems utilized for authentication and data integrity. The text starts by outlining the primary function of signature schemes, which is to offer a digital representation of handwritten signatures or stamped seals. The chapter discusses the fundamental principles of digital

signatures, emphasizing their importance in verifying the validity and preventing denial of digital documents and communications. The conversation then explores many sorts of signature schemes, ranging from conventional ones like RSA to more recent schemes like Elliptic Curve Digital Signature Algorithm (ECDSA) and Digital Signature Algorithm (DSA). The mathematical foundations, operational mechanics, strengths, and potential vulnerabilities of each scheme are examined. The chapter offers a comparative analysis of these designs, emphasizing their suitability in various situations. The practical deployment of signature methods in digital systems is a major concern. An analysis of the protocols and standards for digital signatures, including PKCS#1 for RSA, and their implementation in different technologies and platforms. The chapter examines the legal and regulatory issues of digital signatures, including their recognition and validity in various jurisdictions.

- **Chapter 15, "Lattice-Based Cryptography and NTRU".** This chapter will present the main concepts of lattice-based cryptography and will describe the NTRU encryption system that is included in this branch of cryptography. This chapter provides a detailed examination of lattice-based cryptography, specifically focusing on the NTRU encryption method, a renowned example within this field. The text introduces lattice-based cryptography, discussing its theoretical foundations rooted in the geometry of numbers and its importance as a possible method for constructing cryptosystems that can withstand attacks from quantum computers. The chapter introduces fundamental ideas of lattices in cryptography, such as definitions, properties, and their role in creating complex mathematical problems essential for cryptographic security. The conversation transitions to Nth Degree Truncated Polynomial Ring Units (NTRU), which is among the oldest and most prominent lattice-based encryption techniques. The chapter thoroughly examines NTRU's algorithmic framework, covering key generation, encryption, and decryption procedures. The text delves into the mathematical

complexities of NTRU, focusing on its use of polynomial rings for security, and how this sets it apart from conventional cryptosystems such as RSA and ECC.

- **Chapter 16, "Cryptography Tools".** This chapter discusses cryptography tools that can be used to check the correctness of their implementations, such as CrypTool and OpenSSL.

Conclusion

This first chapter covered the objectives of the book and highlighted the need for cryptography and information security. Then it explained what cryptology, cryptography, and cryptanalysis mean. Finally, the chapter described each upcoming chapter so you can find information of interest quickly.

References

[1] M. I. Aziz and S. Akbar, "Introduction to Cryptography," 2005 International Conference on Microelectronics, Islamabad, Pakistan, 2005, pp. 144-147, doi: 10.1109/ICM.2005.1590056.

[2] Zoubir Z. Mammeri, "Introduction to Cryptography," in Cryptography: Algorithms, Protocols, and Standards for Computer Security , Wiley, 2024, pp.33-58, doi: 10.1002/9781394207510.ch2.

[3] Gómez Pardo, J.L. (2013). Classical Ciphers and Their Cryptanalysis. In: Introduction to Cryptography with Maple. Springer, Berlin, Heidelberg.

[4] Delfs, H., Knebl, H. (2015). Provably Secure Encryption. In: Introduction to Cryptography. Information Security and Cryptography. Springer, Berlin, Heidelberg.

[5] Lorek, P., Yung, M., Zagórski, F. (2024). Mirrored Commitment: Fixing "Randomized Partial Checking" and Applications. In: Pöpper, C., Batina, L. (eds) Applied Cryptography and Network Security. ACNS 2024. Lecture Notes in Computer Science, vol 14585. Springer, Cham.

[6] Sharp, R. (2024). Cryptography. In: Introduction to Cybersecurity. Undergraduate Topics in Computer Science. Springer, Cham.

[7] Damico, T. M. (2009). A brief history of cryptography. Inquiries Journal, 1(11).

[8] Mihailescu, M. I., & Nita, S. L. (2021). Pro Cryptography and Cryptanalysis: Creating Advanced Algorithms with C# and .NET. Apress.

[9] Mihailescu, M. I., & Nita, S. L. (2021). Pro Cryptography and Cryptanalysis with C++ 20: Creating and Programming Advanced Algorithms. Apress.

[10] Mihailescu, M. I., & Nita, S. L. (2021). Cryptography and Cryptanalysis in MATLAB. Apress.

CHAPTER 2

JDK 21: New Features

The Java Platform—developed by Oracle—includes two components: Java Standard Edition (Java SE) and the Java Development Kit (JDK). Java SE is a computing platform that is used to create software applications by developing and deploying portable code for environments such as desktops and servers. Java SE includes different application programming interfaces (APIs) and the Java Class Library. Examples of packages included in Java SE are `java.io`, `java.math`, `java.util`, etc., and it includes APIs such as Applet, AWT, Collections, Swing, JDBC, etc. You can find more about Java SE and its editions at [1].

JDK is the Java development environment, which has two components: Java Language Specification (JLS) and Java Virtual Machine Specification (JVMS). At the moment of writing this book, the latest version of JDK is 21, released on September 19, 2023. Being an LTS version, JDK 21 will receive support and updates until at least September 2031 [2]. Some great resources for Java developers are at [3], which will soon provide a built-in Java compiler to test the examples, and [5], which is the official Oracle documentation for JDK 21. For JDK 17 this will remain the same [4].

In JDK 21, several changes include new features, such as pattern matching for switch (currently under preview), record patterns, the foreign function and memory API, virtual threads, and structured concurrency. You can find the complete list of changes in Java 21 at [6], [7], and [10]. The changes are encoded as "JEP" followed by a number, where JEP means JDK Enhancement Proposals. Some of the main changes are listed here.

The features for JDK 17 can be summarized as follows:

- JEP 453: Structured Concurrency[1]
- JEP 446: Scoped values[2]

[1] JEP 453: Structured Concurrency, `https://openjdk.org/jeps/453`. Last access: 29.02.2024
[2] JEP 446: Scoped values, `https://openjdk.org/jeps/453`. Last access: 29.02.2024

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_2

- JEP 451: Prepare to Disallow the Dynamic Loading of Agents[3]
- JEP 452: Key Encapsulation Mechanism API[4]
- JEP 449: Deprecate the Windows 32-bit x86 Port for Removal[5]
- JEP 445: Unnamed Classes and Instance Main Methods (Preview)[6]
- JEP 443: Unnamed Patterns and Variables (Preview)[7]
- JEP 439: Generational ZGC[8]
- JEP 440: Record Patterns[9]
- JEP 433: Pattern Matching for switch (Fourth Preview)[10]
- JEP 448: Vector API (Sixth Incubator)[11]
- JEP 442: Foreign Function & Memory API (Third Preview)[12]
- JEP 444: Virtual Threads[13]
- JEP 431: Sequenced Collections[14]
- JEP 430: String Templates (Preview)[15]

[3] JEP 451: Prepare to Disallow the Dynamic Loading of Agents, https://openjdk.org/jeps/451. Last access: 29.02.2024

[4] JEP 452: Key Encapsulation Mechanism API, https://openjdk.org/jeps/452. Last access: 29.02.2024

[5] JEP 449: Deprecate the Windows 32-bit x86 Port for Removal, https://openjdk.org/jeps/449. Last access: 29.02.2024

[6] JEP 445: Unnamed Classes and Instance Main Methods (Preview), https://openjdk.org/jeps/445. Last access: 29.02.2024

[7] JEP 443: Unnamed Patterns and Variables (Preview), https://openjdk.org/jeps/443. Last access: 29.02.2024

[8] JEP 439: Generational ZGC, https://openjdk.org/jeps/439. Last access: 29.02.2024

[9] JEP 440: Record Patterns, https://openjdk.org/jeps/440. Last access: 29.02.2024

[10] JEP 433: Pattern Matching for switch (Fourth Preview), https://openjdk.org/jeps/433. Last access: 29.02.2024

[11] JEP 448: Vector API (Sixth Incubator), https://openjdk.org/jeps/448. Last access: 29.02.2024

[12] JEP 442: Foreign Function & Memory API (Third Preview), https://openjdk.org/jeps/442. Last access: 29.02.2024

[13] JEP 444: Virtual Threads, https://openjdk.org/jeps/444. Last access: 29.02.2024

[14] JEP 431: Sequenced Collections, https://openjdk.org/jeps/431. Last access: 29.02.2024

[15] JEP 430: String Templates (Preview), https://openjdk.org/jeps/430. Last access: 29.02.2024

Note that, in the following examples, the Eclipse environment should be configured for use with JDK 21. We'll now elaborate on the JEPs of interest to the field of cryptography. Java JDK 21 brings new features and updates to enhance the performance, functionality, and reliability of the Java platform. Here is a synopsis of the main features:

- **JEP 433: Pattern Matching for switch (Fourth Preview).** This feature improves the Java language by adding pattern matching for `switch` expressions and statements, making it easier to use patterns in `switch`. Developers can utilize patterns within `switch` labels to create code that is more expressive and adaptable.
- **JEP 440: Record Patterns.** This preview introduces record patterns in Java, enhancing data processing capabilities with sophistication and conciseness. Record patterns can break down record values, offering a more straightforward method to manage intricate data structures.
- **JEP 442: Foreign Function & Memory API (Third Preview).** This feature offers a Java API for invoking native code and managing native memory. The goal is to substitute the Java Native Interface (JNI) with a more efficient and dependable option for interacting with non-Java programs.
- **JEP 444: Virtual Threads.** Virtual threads are efficient threads that significantly decrease the work involved in creating, managing, and monitoring high-throughput concurrent applications. Their goal is to simplify the management and scalability of concurrency in Java programs.
- **JEP 453: Structured Concurrency.** Structured Concurrency is a new feature in Java that offers a simpler and more dependable approach to concurrent programming. It seeks to enhance the clarity and dependability of multithreaded Java applications by a methodical approach to concurrency.

The improvements are designed to enhance Java's capabilities for modern application development, making it more powerful and user-friendly for developers. Features labeled as "Preview" are subject to change before being officially included in the mainstream Java language, while those labeled as "Incubator" are experimental features designed to collect input.

These new features and enhancements improve the Java platform's performance, usability, and security. Next you'll find more details and an example of each.

JEP 433: Pattern Matching for switch (Fourth Preview)

This feature enhances the Java language with pattern matching for the `switch` expression and statement, allowing for more expressive and safe code.

Example:

```
java
// before JDK 21
String formatted = switch (obj) {
    case Integer i -> String.format("int %d", i);
    case Long l    -> String.format("long %d", l);
    case Double d  -> String.format("double %f", d);
    default        -> obj.toString();
};

// starting with JDK 21
String formatted = switch (obj) {
    case Integer i -> "int %d".formatted(i);
    case Long l    -> "long %d".formatted(l);
    case Double d  -> "double %f".formatted(d);
    default        -> obj.toString();
};
```

JEP 440: Record Patterns

Record patterns simplify the deconstruction and processing of record values in pattern matching operations.

Example:

```
java
record Point(int x, int y) {}

// applying Record Patterns
boolean isOrigin(Object obj) {
    return switch (obj) {
        case Point(int x, int y) when x == 0 && y == 0 -> true;
        default -> false;
    };
}
```

JEP 442: Foreign Function & Memory API (Third Preview)

This API provides a pure Java API for calling native code and working with native memory. It's part of Project Panama and aims to replace the JNI for improved performance and usability.

Example:

```
java
// how to access a C library function
try (var scope = ResourceScope.newConfinedScope()) {
    var path = CLinker.systemCLinker().lookup("ACLibraryFunction").get();
    var function = CLinker.systemCLinker().downcallHandle(
        path,
        MethodType.methodType(void.class),
        FunctionDescriptor.ofVoid()
    );
    function.invokeExact();
}
```

JEP 444: Virtual Threads

Virtual threads are lightweight threads that dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications.

Example:

```
java
// declaring a virtual thread
Thread.startVirtualThread(() -> {
    System.out.println("This is a virtual thread.");
});
```

JEP 409[16]: Sealed Types

Sealed types were introduced in previous versions, but JDK 21 enhances them further, allowing for more control over subclassing in a domain-specific manner.

[16] JEP 409, https://openjdk.org/jeps/409. Last access: 02.03.2024. It is the same in JDK 21 as in JDK 17. No improvements have been made so far.

Example:

```
java
// how to define a sealed class
public sealed class Shape permits Circle, Square {
    // class body
}

final class Circle extends Shape {
    // class body
}

final class Square extends Shape {
    // class body
}
```

Each of these features represents important improvements to the Java ecosystem, making the language more expressive, efficient, and safe for developers. The examples provided here illustrate how each feature can be utilized in Java code.

Integrating the new Java JDK 21 features into the context of cryptography, we can explore how these enhancements could potentially streamline cryptographic operations and code handling. The following are adapted examples of the features, focusing on cryptographic scenarios.

JEP 433: Pattern Matching for switch (Fourth Preview)

Use case: Handling different cryptographic algorithms based on object types.

Example:

```
java

// Consider it as an instance of various cipher classes
Object algorithm = getAlgorithmObject();

String algorithmInfo = switch (algorithm) {
    case AESCipher c -> "AES algorithm with key size " + c.keySize();
    case RSACipher c -> "RSA algorithm with modulus size " +
    c.modulusSize();
    case DESCipher c -> "DES algorithm";
    default -> "Unknown algorithm";
};
```

JEP 440: Record Patterns

Use case: Extracting components of a cryptographic key or a structured message.

Example:

```
java
record KeyPair(PublicKey publicKey, PrivateKey privateKey) {}

boolean isKeyPairValid(Object obj) {
    return switch (obj) {
        case KeyPair(PublicKey pub, PrivateKey priv) when pub.
        getAlgorithm().equals(priv.getAlgorithm()) -> true;
        default -> false;
    };
}
```

JEP 442: Foreign Function & Memory API (Third Preview)

Use case: Interacting with native cryptographic libraries for operations not available or efficient in Java.

Example:

```
java

// using a native library function to perform a cryptographic operation,
e.g., hashing
try (var scope = ResourceScope.newConfinedScope()) {
    var path = CLinker.systemCLinker().lookup("native_hash_
    function").get();
    var hashFunction = CLinker.systemCLinker().downcallHandle(
        path,
        MethodType.methodType(void.class, byte[].class),
        FunctionDescriptor.ofVoid(CLinker.C_POINTER)
    );
    byte[] data = "example data".getBytes(StandardCharsets.UTF_8);
    hashFunction.invokeExact(data);
}
```

JEP 444: Virtual Threads

Use case: Handling multiple cryptographic operations in parallel without the overhead of traditional threads.

Example:

```
java

// Executing cryptographic operations in parallel using virtual threads
List<String> messages = List.of("Message #1", "Message #2", "Message #3");
messages.parallelStream().forEach(message ->
    Thread.startVirtualThread(() -> {
        // consider for a cryptographic operation, e.g., encryption
        System.out.println("Encrypting: " + message);
    })
);
```

JEP 409: Sealed Types

Use case: Defining a hierarchy of cryptographic operations to ensure type safety and clarity.

Example:

```
java
// Defining a sealed class hierarchy for cryptographic operations
public sealed class CryptoOperation permits Encrypt, Decrypt {
    // class body
}

final class Encrypt extends CryptoOperation {
    // class body, e.g., method for encryption
}

final class Decrypt extends CryptoOperation {
    // class body, e.g., method for decryption
}
```

In these examples, we saw how the new features in Java JDK 21 could be employed to enhance cryptographic operations, making the code more expressive, type-safe, and efficient.

In Listing 2-1 are printed available PRNGs with some information about them: their type (arbitrary jumpable, hardware, jumpable, leapable, splittable, statistical, stochastic, or streamable) and the number of bits used to maintain the state of the seed. After that, lines 28 and 29 show how PRNGs are instantiated in the new Java 21, while line 31 shows the old instantiation of the `Random` class. Lines 33, 34, and 36 show the default representation for the new implementations and the shortcut for the default representation, respectively. Listing 2-1 shows the results.

Listing 2-1. Using PRNGs in Java 21

```
1   import java.util.Random;
2   import java.util.random.RandomGenerator;
3   import java.util.random.RandomGeneratorFactory;
4   import java.util.stream.Stream;
5
6   public class JEP356Example {
7
8         public static void main(String[] args) {
9
10              Stream<RandomGeneratorFactory<RandomGenerator>> allPRNGs =
                RandomGeneratorFactory.all();
11              allPRNGs.map(prng -> prng.name() + " [ Group: "  + prng.
                group() + "; "
12                          + (prng.isArbitrarilyJumpable() ? " arbitrary-
                            jump" : "")
13                          + (prng.isHardware()? " hardware" : "")
14                          + (prng.isJumpable() ? " jump" : "")
15                          + (prng.isLeapable()? " leap" : "")
16                          + (prng.isSplittable() ? " split" : "")
17                          + (prng.isStatistical()? " statistical" : "")
18                          + (prng.isStochastic()? " stochastic" : "")
19                          + (prng.isStreamable() ? " stream" : "")
20                          + "; noOfBits: "+ prng.stateBits()
21                          + "]"
22                          ).sorted().forEach(System.out::println);
23
```

```
24              System.out.println("\n*****\n");
25
26              RandomGenerator prng1 = RandomGeneratorFactory.
                of("Random").create(45);
27              System.out.println("prng1 - " + prng1.getClass());
28              RandomGenerator prng2 = new Random(45);
29              System.out.println("prng2 - " + prng2.getClass());
30              RandomGenerator prng3 = RandomGeneratorFactory.
                getDefault().create(45);
31              System.out.println("prng3 - " + prng3.getClass());
32              RandomGenerator prng4 = RandomGenerator.getDefault();
33              System.out.println("prng4 - " + prng4.getClass());
34          }
35  }
```

In Figure 2-1, the `Legacy` category is for the old implementation of PRNGs, while `LXM` and `Xoroshiro` are newly introduced. Note that the new implementations are not thread-safe; namely, the same instance can be shared between more threads (this is not a common practice while developing applications). Instead, `java.util.Random` and `java.security.SecureRandom` are thread-safe.

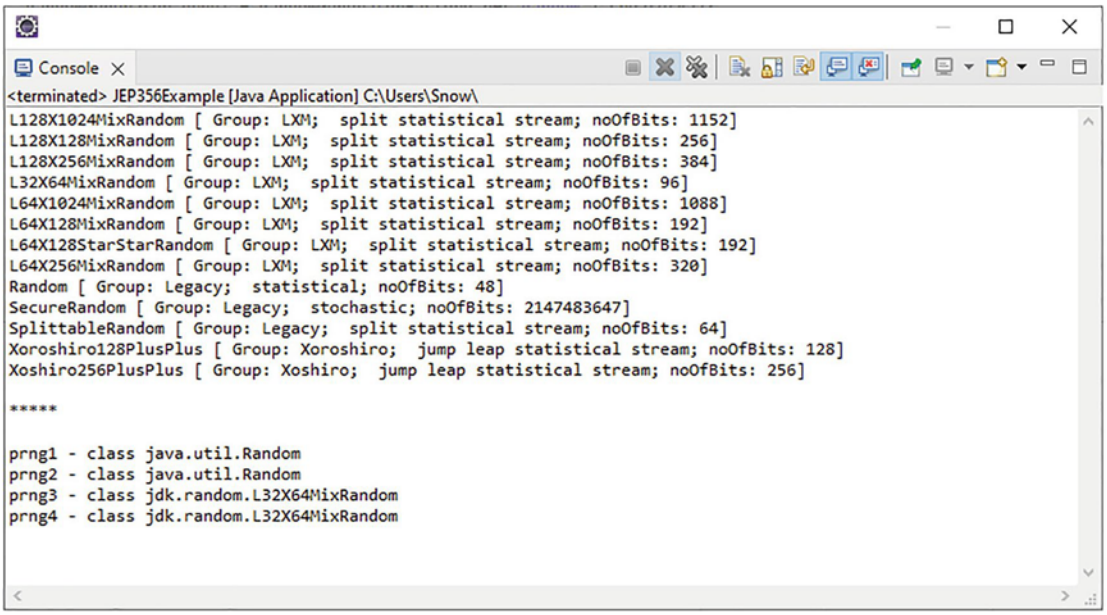

Console
<terminated> JEP356Example [Java Application] C:\Users\Snow\
L128X1024MixRandom [Group: LXM; split statistical stream; noOfBits: 1152]
L128X128MixRandom [Group: LXM; split statistical stream; noOfBits: 256]
L128X256MixRandom [Group: LXM; split statistical stream; noOfBits: 384]
L32X64MixRandom [Group: LXM; split statistical stream; noOfBits: 96]
L64X1024MixRandom [Group: LXM; split statistical stream; noOfBits: 1088]
L64X128MixRandom [Group: LXM; split statistical stream; noOfBits: 192]
L64X128StarStarRandom [Group: LXM; split statistical stream; noOfBits: 192]
L64X256MixRandom [Group: LXM; split statistical stream; noOfBits: 320]
Random [Group: Legacy; statistical; noOfBits: 48]
SecureRandom [Group: Legacy; stochastic; noOfBits: 2147483647]
SplittableRandom [Group: Legacy; split statistical stream; noOfBits: 64]
Xoroshiro128PlusPlus [Group: Xoroshiro; jump leap statistical stream; noOfBits: 128]
Xoshiro256PlusPlus [Group: Xoshiro; jump leap statistical stream; noOfBits: 256]
*****
prng1 - class java.util.Random
prng2 - class java.util.Random
prng3 - class jdk.random.L32X64MixRandom
prng4 - class jdk.random.L32X64MixRandom

Figure 2-1. *The result of Listing 2-1*

JEP 452. Key Encapsulation Mechanism API. This is one of the features that enhances the security of the JDK, as the internal APIs cannot be accessed with the `--illegal-access` option at the command line anymore; this is because in JDK 21 all internal elements are now strongly encapsulated. The exceptions are crucial APIs, such as `sun.misc.Unsafe`. The purpose of this feature is to encourage developers to use standard APIs instead of internal APIs. This change increases the security and the maintainability of the JDK. On the official page of this JEP, the developers give some examples of pieces of code that will not work anymore in JDK 21 and provide explanations for each example. You can learn more about JEP 452 at [10].

JEP 433: Pattern Matching for switch (Fourth Preview). By far, using pattern matching in `switch` expressions and statements will be easier and more elegant. In addition, this will save lots of lines of code. Listing 2-2 is an example of pattern matching for `switch` that can be used in cryptography. Consider three classes that implement the RSA, AES, and ElGamal algorithms (lines 1–11); the user should introduce only the secret key or the pair of the secret key and public key corresponding to the chosen encryption system's type. Note that for the following example, the usage of preview language features should be enabled in order to work properly.

Listing 2-2. Example of Pattern Matching for switch

```
class RSA {

}

class AES {

}

class ElGamal {

}

public class JEP406Example {

      static void TypeOfKeys(Object o) {
            if (o instanceof RSA ) {
                System.out.println("Two keys needed. Type the secret
                on the first line. Type the public key on the second
                line.");
            } else if (o instanceof AES) {
                System.out.println("One key needed. Type the secret
                key on the first line.");
            } else if (o instanceof ElGamal) {
                System.out.println("Two keys needed. Type the secret
                on the first line. Type the public key on the second
                line.");
            }
      }

      static void TypeOfKeysPatternSwitch(Object o) {
            if(o == null)
               throw new NullPointerException();
            else
            {

```

```
30                    switch (o.getClass().toString()) {
31                            case null -> throw new
                              NullPointerException();
32                            case "RSA2" -> System.out.println("Two keys
                              needed. Type the secret on the first line.
                              Type the public key on the second line.");
33                            case "AES2" -> System.out.println("One key
                              needed. Type the secret key on the first
                              line.");
34                            case "ElGamal2" -> System.out.println
                              ("Two keys needed. Type the secret on the
                              first line. Type the public key on the
                              second line.");
35                            default -> System.out.println("Pick an
                              encryption system");
36                    }
37              }
38        }
39
40        public static void main(String[] args) {
41              TypeOfKeys(new RSA());
42              TypeOfKeysPatternSwitch(new RSA());
43
44              System.out.println("\n***\n");
45
46              TypeOfKeys(new AES());
47              TypeOfKeysPatternSwitch(new AES());
48        }
49  }
```

In Listing 2-2, two approaches were used to print the message for the user: the regular one, in the function `TypeOfKeys` (lines 16–27), and the version with pattern matching for the `switch` statement (lines 29–45). Note that the result should be the same for the same type of object that is passed to the two functions. You can check the result in Figure 2-2, and you can learn more about JEP 433 at [11].

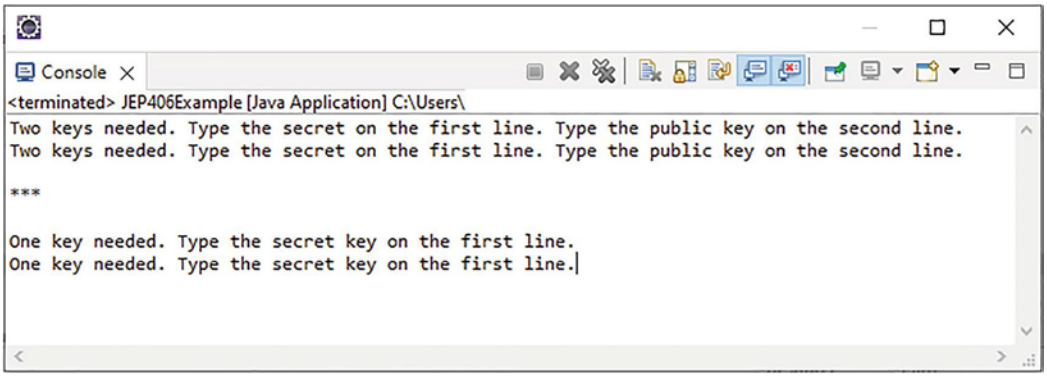

Console
<terminated> JEP406Example [Java Application] C:\Users\
Two keys needed. Type the secret on the first line. Type the public key on the second line.
Two keys needed. Type the secret on the first line. Type the public key on the second line.
***
One key needed. Type the secret key on the first line.
One key needed. Type the secret key on the first line.

Figure 2-2. *The result of using pattern matching for switch*

JEP409, "Sealed Classes." These types of classes are important in object-oriented programming because they can restrict the list of classes or interfaces that are extensions or implementations of them, giving more control to the code that implements them. For example, consider there is a fixed collection of encryption systems stored within an enum. As there is a fixed number of encryption systems, a default clause wouldn't be necessary when switching between them; see Listing 2-3.

Listing 2-3. Sketch for Cryptosystems Stored as enum

```
enum Cryptosystem { AES, RSA, ElGamal }
Cryptosystem cipher = ...
switch (cipher) {
  case AES: ...
  case RSA: ...
  case ElGamal: ...
}
```

In situations in which a given number of kinds of values should be managed (instead of managing a set of values), a hierarchy of classes can be used for listing these kinds of values. For the previous enum example, this would be transformed into Listing 2-4.

Listing 2-4. Converting the enum Example to a Class Hierarchy

```
interface Cryptosystem { ... }
final class AES implements Cryptosystem { ... }
final class RSA implements Cryptosystem { ... }
final class ElGamal implements Cryptosystem { ... }
```

However, superclasses should be widely accessible but not widely extensible. Sealed classes were introduced to give more control over how extensibility and implementation are used. The previous example can be modified by using sealed classes, as shown in Listing 2-5.

Listing 2-5. Sealed Classes

```
1   public class JEP409Example {
2
3         static void Test(Cryptosystem c) {
4               if (c instanceof AES_Cryptosystem)
5                     System.out.println("AES chosen");
6               else
7                     if (c instanceof RSA_Cryptosystem)
8                           System.out.println("RSA chosen");
9                     else
10                          throw new RuntimeException("Unknown instance
                            of Cryptosystem.");
11
12        }
13        public static void main(String[] args) {
14              AES_Cryptosystem aes = new AES_Cryptosystem();
15              RSA_Cryptosystem rsa = new RSA_Cryptosystem();
16
17              Test(aes);
18              Test(rsa);
19        }
20  }
```

In Listing 2-5, three classes are used: `Cryptosystem`, `AES_Cryptosystemn`, and `RSA_Cryptosystem`. Note that to use them as sealed classes and classes that extend the sealed class, each class must be defined in its own file. To prevent further extensions, the classes that extend the sealed class should be declared as `final`. Here is the definition for the three classes used in the example:

```
public abstract sealed class Cryptosystem
    permits AES_Cryptosystem, RSA_Cryptosystem {
      String encryptMessage() { return null;}
      String decryptMessage() { return null; }
}
public final class AES_Cryptosystem extends Cryptosystem {
}
public final class RSA_Cryptosystem extends Cryptosystem {
}
```

You can find more examples of sealed classes at [12].

JEP412, "Foreign Function & Memory API (Incubator)." With this new feature, Java code can work with foreign code and data from outside of the Java runtime, other than C. In cryptography, this can be useful, especially when working with mathematical tools. For example, Matlab code can be used for mathematical functions or algorithms that are already implemented, such as statistical functions. Using the foreign code is safe and efficient. You can learn more about JEP412 at [13].

JEP414, "Vector API (Second Incubator)." Using this feature achieves better performance and implementation for vector instructions compared to equivalent scalar computations. The improved API provides features such as translating byte vectors to Boolean arrays, or vice versa. You can learn more about JEP412 at [16].

JEP415, "Context-Specific Deserialization Filters." This is another example of a feature that is great for cryptographic purposes. The motivation behind this feature is that data that is received from different sources is untrusted because in the world of the Internet, often the source is unknown, untrusted, or unauthenticated. Therefore, when untrusted data is deserialized, it can be dangerous. For example, an attacker can exploit the vulnerabilities behind serialized/deserialized data injecting malicious behavior. With this feature, a filter factory for Java Virtual Machine (JVM) was introduced that can be configured. When it is instantiated as an object of the `ObjectInputStream` type, the JVM filter factory is invoked, and the result is used to initialize each filter per stream. You can learn more about JEP415 at [18].

Conclusion

This chapter presented the main features added in Java 21. By far one of the most exciting added features is the new pseudorandom generator implementations and pattern matching for `switch` instructions. The chapter presented some use cases for cryptography.

Java JDK 21 introduces powerful new capabilities and changes that greatly benefit the Java ecosystem, especially for cryptographic applications. Features like pattern matching for `switch`, record patterns, the foreign function and memory API, virtual threads, and sealed types provide Java developers with more expressive, efficient, and type-safe methods to execute intricate cryptographic operations and patterns.

Utilizing pattern matching for `switch` enables developers to create more succinct and legible code for managing different cryptographic objects and hence improves code maintainability and decreases the chances of errors.

Record patterns simplify the deconstruction of intricate data types like cryptographic keys or structured communications, making it easier to handle such data in a clear and logical manner.

The foreign function and memory API allows Java programs to utilize native cryptography libraries efficiently and without the need for JNI, giving access to optimized, platform-specific code.

Virtual threads present a streamlined concurrency approach that is advantageous for executing numerous cryptographic tasks simultaneously, enhancing application efficiency without the complexities of conventional thread administration.

Sealed types offer a methodical approach to specifying cryptographic procedures, guaranteeing that they are both expandable and safe, while following established hierarchies and limitations.

Java JDK 21's new features aim to improve the Java programming language's capabilities in cryptography by providing additional tools and constructs for developing safe, efficient, and clean code. Java's continuous progress aligns with contemporary development requirements, solidifying its status as a resilient, adaptable, and future-oriented programming language suitable for cryptography and general-purpose applications.

References

[1] Java SE at a Glance, https://www.oracle.com/ro/java/technologies/java-se-glance.html. Last access: 29.02.2024

[2] OpenJDK, JDK 21, https://openjdk.org/projects/jdk/21/. Last access: 29.02.2024

[3] Dev. Java: The destination for Java developer, https://dev.java/. Last access: 29.02.2024

[4] JDK 17 Documentation, https://docs.oracle.com/en/java/javase/17/. Last access: 29.02.2024

[5] JDK 21 Documentation, https://docs.oracle.com/en/java/javase/21/. Last access: 29.02.2024

[6] JDK 17 Release Notes, https://www.oracle.com/java/technologies/javase/17-relnote-issues.html. Last access: 29.02.2024

[7] JDK 21 Release Notes, https://www.oracle.com/java/technologies/javase/21-relnote-issues.html. Last access: 29.02.2024

[8] Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification, https://docs.oracle.com/en/java/javase/17/docs/api/index.html.

[9] JDK 17.0.1 General-Availability Release[17], https://jdk.java.net/17/. Last access: 29.02.2024

[10] JEP 452: Key Encapsulation Mechanism API, https://openjdk.org/jeps/452. Last access: 29.02.2024

[11] JEP 433: Pattern Matching for switch (Fourth Preview), https://openjdk.org/jeps/433. Last access: 29.02.2024

[17] JDK 17 Releases: "JDK 17 has been superseded. Please visit jdk.java.net for the current version. Older releases, which do not include the most up to date security vulnerability fixes and are no longer recommended"; https://jdk.java.net/17/.

[12] JEP 356: Enhanced Pseudorandom Number Generators, https://openjdk.java.net/jeps/356.

[13] JEP 403: Strongly Encapsulate JDK Internals, https://openjdk.java.net/jeps/403.

[14] JEP 406: Pattern Matching for switch (Preview), https://openjdk.java.net/jeps/406.

[15] JEP 409: Sealed Classes, https://openjdk.java.net/jeps/409.

[16] JEP 412: Foreign Function & Memory API (Incubator), https://openjdk.java.net/jeps/412.

[17] JEP 414: Vector API (Second Incubator), https://openjdk.java.net/jeps/414.

[18] JEP 415: Context-Specific Deserialization Filters, https://openjdk.java.net/jeps/415.

CHAPTER 3

Roadmap and Vision for Jakarta EE 11

Jakarta EE 11 will represent the first major release of Jakarta EE, and its goal is to deliver a specific set of specifications for Jakarta EE technologies.

By late 2021, the following key themes have been proposed by the community for shaping Jakarta EE 11:

- Aligning with Java Standard Edition (SE)
- Filling the gaps in the standardization
- Aligning with contexts and dependency injection (CDI)

Starting with Java EE 8, Jakarta Security is a new API with three authentication procedures: Basic, Form, and an extension of Form that perfectly suits the process of implementation within Java Server Faces (JSF).

With Jakarta EE 11, the goal is to add new authentication mechanisms. The first mechanism on the list is represented by Client-Cert and Digest. In this way, Jakarta Security will provide a total replacement of the authentication mechanisms that are supported by Java Servlet. Another goal is to offer support for OpenID, OAuth, and JSON Web Token (JWT). JWT is an interesting topic that is currently under discussion due to the process of adding it to the MicroProfile project.

The plan for Jakarta EE 11 is to progress Java for cloud-native application development, improving the platform's significance and flexibility in contemporary enterprise settings. Jakarta EE, managed by the Eclipse Foundation, is advancing from Java EE by emphasizing open standards, community-driven development, and innovation in enterprise Java. The primary parts of the plan and vision for Jakarta EE 11 are as follows:

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_3

- **Support for cloud-native technologies.** Jakarta EE 11 seeks to enhance its backing for cloud-native development paradigms, focusing on microservices architectures, containerization, and orchestration using Kubernetes. The aim is to simplify the creation, implementation, and control of scalable and robust Java applications in cloud settings.
- **Improved modularity and microservices.** The upcoming release aims to streamline microservices development by improving modularity, building upon the foundation set by Jakarta EE 9 and 10. This may include enhanced project structures, better interface with microservices frameworks, and functionalities facilitating autonomous development, testing, and deployment of services.
- **Enhanced performance and efficiency.** Jakarta EE 11 is anticipated to bring advancements that aim to decrease resource usage, enhance startup times, and boost throughput, while performance optimizations and efficiency improvements are consistently prioritized. These enhancements are essential for apps operating in cloud settings, as resource usage directly affects expenses.
- **Compatibility with Java SE features.** Jakarta EE 11 aims to synchronize more closely with new and planned capabilities in the Java language and platform, including record types, pattern matching, and other language improvements. This alignment allows developers to utilize the most recent Java capabilities in their enterprise applications, enhancing the codebase's expressiveness and efficiency.
- **Improved security measures.** Jakarta EE 11 is anticipated to bring additional security features and improvements, with an emphasis on OAuth 2.0, OpenID Connect, and enhanced support for securing microservices, in response to the critical importance of security in enterprise applications. The improvements will assist developers in securing apps and services more efficiently within a cloud-native environment.

- **Criteria and consistency.** Jakarta EE 11 will maintain backward compatibility with previous versions and adhere to new standards in web technologies, data access, messaging, and other areas, demonstrating its dedication to standards and compatibility. By doing this, applications developed on Jakarta EE are guaranteed to be both portable and resistant to future changes.
- **Involvement of community and ecosystem.** The roadmap highlights the significance of community and ecosystem involvement, with the Eclipse Foundation promoting contributions from the community, cooperation with other open-source projects, and alliances with technology suppliers. The collaborative approach seeks to promote innovation and cater to the changing requirements of corporate Java developers.
- **Future vision.** The goal for Jakarta EE 11 and beyond is to position Jakarta EE as the leading platform for creating and launching cloud-native enterprise Java applications. Jakarta EE aspires to stay current and competitive in the fast-changing technological field by adopting modern development practices, improving performance and scalability, and nurturing a lively community.

Jakarta EE 11 aims to strategically advance enterprise Java to meet the current and future requirements of application development and deployment in the cloud. Jakarta EE is prepared to address the demands and possibilities of contemporary enterprise computing by emphasizing cloud-native features, community involvement, and ongoing innovation.

Security is a fundamental aspect of Jakarta EE 11, meeting the increasing need for strong, adaptable, and all-encompassing security measures in corporate and cloud-native applications. Jakarta EE 11 aims to improve the security of Java applications by focusing on key areas to address the changing threat landscape and the requirements of contemporary application development techniques. The following are the security improvements and main focuses of Jakarta EE 11:

- **Improved Authentication and Authorization.** Jakarta EE 11 intends to enhance support for contemporary authentication and authorization methods like OAuth 2.0 and OpenID Connect. These protocols are essential for safeguarding RESTful services and microservices architectures by offering a standardized method for applications to request and handle access tokens.

- **Token-based authentication.** Prioritizing token-based authentication procedures aids in creating stateless apps, crucial for scalability in cloud environments.
- **Encrypted communication.** Enhancements have been made to Transport Layer Security (TLS) configuration and management to enable secure communication routes. This involves simplifying the setup of SSL/TLS for online apps and handling certificates in a cloud-native setting. It is vital to ensure that secure communication via the HTTP/2 protocol is fully supported and optimized in Jakarta EE 11 due to its increasing prevalence.
- **Ensure the safe implementation and setup.** Jakarta EE 11 seeks to establish secure-by-default settings to reduce setup errors that may result in security vulnerabilities. This encompasses predefined settings for secure session handling, protection against Cross-Site Request Forgery (CSRF), and policies for Cross-Origin Resource Sharing (CORS). Enhancements have been made in the management of security configurations, especially for applications in cloud environments, to ensure that security settings can be handled and inspected quickly and reliably. CSRF is a form of attack that compels a user to perform unauthorized actions on a web application while they are logged in. A CSRF attack can lead to the transmission of illegal commands by a user who has trust in the online application. To effectively avoid such attacks, it is vital to understand how they function for ethical and educational reasons. This is a fictional example demonstrating how a CSRF attack could be developed, aimed at understanding its mechanics and promoting the implementation of countermeasures against such vulnerabilities. Performing CSRF attacks without permission is both illegal and unethical. Let's imagine a simple Java web application that gives the possibility to the users to change their password via a form submission. The form might look like this in HTML:

```
        <!—A simple form of an web application when the user is
        authenticated with -->
        <form action="https://www.mariusmihailescu.com/changeYourPassword"
        method="POST">
            <input type="password" name="newPassword" />
            <input type="submit" value="Change Password" />
        </form>
```

The corresponding Java back end (simplified for illustration) might handle the form submission as follows:

```
// This is a simplified servlet example handling the password
change request
@WebServlet("/changeYourPassword")
public class ChangePasswordServlet extends HttpServlet
{
    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException
{
        String newPassword = request.getParameter("newPassword");
        // We suppose that getUserFromSession is a method that retrieves
        the user based on the current session
        User user = getUserFromSession(request.getSession());

        if (user != null) {
            // change the user's password (it's a must)
            changeUserPassword(user, newPassword);
            response.sendRedirect("/passwordChangedSuccessfully.jsp");
        } else {
            response.sendRedirect("/errorPage.jsp");
        }
    }

    // Simulating with a dummy method to illustrate getting a user object
    from session
    private User getUserFromSession(HttpSession session) {
        // Implementation to retrieve user object from session
```

```
        return new User(); // a simple call for this example
    }

    // Declare a dummy method to demonstrate changing the user's password
    private void changeUserPassword(User user, String newPassword) {
        // Implementation for changing the user's password
    }

    // Dummy user class
    class User {
        // User details
    }
}
```

CSRF vulnerability. A vulnerability arises because the application does not verify whether the incoming request originated from a trusted source. The attacker has the possibility to craft a malicious website with a form that auto-submits a request to the `changePassword` endpoint when visited by the user:

```
<!-- Malicious website -->
<form action="http://www.mariusmihailescu.com/changePassword" method="POST"
id="maliciousForm">
    <input type="hidden" name="newPassword" value="attackerPassword" />
</form>
<script>document.getElementById('maliciousForm').submit();</script>
```

If an authenticated user visits this malicious page, the form would automatically submit to the legitimate site's `changePassword` endpoint, changing the user's password without their knowledge or consent.

Preventing CSRF attacks. To prevent CSRF attacks, it's crucial to implement anti-CSRF tokens in the application. A unique token is generated by the server for each user session and included as a hidden field in the form. The server then verifies this token when processing form submissions to ensure the request is legitimate. Java web frameworks, such as Spring Security, offer built-in support for CSRF protection by automatically including and verifying CSRF tokens in forms. Remember, understanding and preventing CSRF vulnerabilities is a critical aspect of secure web application development. Always follow best practices and utilize security frameworks to protect your applications.

- **Security of microservices.** Service-to-service security is crucial in a microservices architecture for ensuring secure communication between services. Jakarta EE 11 emphasizes offering patterns and techniques for secure service-to-service communication, such as mutual TLS, JWT, and backing for service meshes that can handle security issues. Identity propagation represents the action of ensuring that the identity of the requester may be securely transferred through and validated by different services in a request chain in microservices architectures. JWT is a concise and secure method of conveying assertions between two entities. Java has several libraries for handling JWTs, including `jjwt`[1] by `JFrog`, a widely used option. Let's take a look at an example for the `jjwt` library. This example will include the following steps:
  - **Creating the JWT.** Signing the token using a secret key.
  - **Parsing the JWT.** Verifying the signare and extracting the claim(s).

  Step 1. Add the `jjwt` dependency.
  First, you need to add the `jjwt` library to your project. If you're using Maven, include the following in your `pom.xml`:

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

  For Gradlke, we need to add this to our `build.gradle`:

```
implementation 'io.jsonwebtoken:jjwt:0.9.1'
```

[1] jjwt library, https://github.com/jwtk/jjwt. Last access: 02.03.2024

Step 2. Look at a code example. This represents a basic example that will demonstrate how we can create and parse a JWT token:

```
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.SignatureAlgorithm;
import java.util.Date;

public class StefaniaMarius_Apress_JWTExample {
    private static final String SECRET_KEY =
    "aRandomSecretKey";

    public static void main(String[] args) {
        // Create a JWT Token
        String jws = Jwts.builder()
                .setSubject("marius2024") // a user ID
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.
                currentTimeMillis() + 3600000)) //1 hour
                expiration
                .signWith(SignatureAlgorithm.HS256,
                SECRET_KEY)
                .compact();

        System.out.println("The JWT Token is: " + jws);

        // Parse the JWT Token
        Claims claims = Jwts.parser()
                .setSigningKey(SECRET_KEY)
                .parseClaimsJws(jws)
                .getBody();

        System.out.println("Subject: " + claims.
        getSubject());
        System.out.println("Issued at: " + claims.
        getIssuedAt());
    }
}
```

- **Ongoing security.** This example highlights the significance of an efficient approach for implementing security patches and upgrades on the Jakarta EE platform and its associated applications to reduce the vulnerability period. It encourages and supports proactive security testing inside the development life cycle by integrating security testing tools and services to find and fix vulnerabilities early.
- **Community engagement and standards participation.** Engaging with security groups and standards bodies to stay updated on evolving security risks and best practices helps Jakarta EE maintain a leading position in corporate application security. The open security model represents the action of encouraging community contributions to security features and documentation to promote a collaborative approach to safeguarding enterprise Java applications.

Ultimately, Jakarta EE 11's security strategy is thorough, encompassing authentication, secure communication, configuration, microservices security, and ongoing security protocols. Jakarta EE 11 intends to equip developers with the necessary tools and frameworks to create safe, resilient, and scalable applications tailored for the cloud-native environment, thereby solidifying Java's position as a preferred option for enterprise application development.

Jakarta Security supports different annotations, such as `@RolesAllowed` and `@RunAs`, with support for further extensions. At the moment, `@RolesAllowed` is supported only by EJB in Jakarta EE. The annotation will throw an exception if access is denied by a bean method.

Related to the authentication mechanisms, `@RolesAllows` will trigger an authentication mechanism when access is denied by a Jakarta REST resource method.

The Jakarta EE 11 platform will include a serious set of components; Table 3-1 shows their status.

Table 3-1. *Jakarta EE 11 Web Profile*[2]

Component	Status
Authentication 3.0	Updated
Concurrency 3.0	Updated
CDI 4.0	Updated
Expressions Language 5.0	Updated
Faces 4.0	Updated
Security 3.0	Updated
Servlet 6.0	Updated
Standard Tag Libraries 3.0	Updated
Persistence 3.1	Updated
Server Pages 3.1	Updated
WebSocket 2.1	Updated
Bean Validation 3.0	Not updated
Debugging Support 2.0	Not updated
Enterprise Beans Lite 4.0	Not updated
Managed Beans 2.0	Not updated
Transactions 2.0	Not updated
RESTful Web Services 3.1	Updated
JSON Processing 2.1	Updated
JSON Binding 2.1	Updated
Annotations 2.0	Not updated
Interceptors 2.0	Not updated
Dependency Injection 2.0	Not updated
CDI Lite 4.0	Updated/new
Config 1.0	New

[2] Is based on Jakarta EE 10; currently Jakarta EE 11 is under review and development. Once it is finished, the new web profile features will be found here: https://jakarta.ee/specifications/webprofile/11/.

User-friendly authentication modules are one of the most important features handled by Jakarta Security within Jakarta EE 11; they enable authorization rules that can be easily implemented.

These authorization modules are designed to work as low-level, portable mechanisms. The steps used to develop and install these modules have become more stable, making them more useful for business applications.

The specifications that are available with the Jakarta EE 11 platform and Web Profile for business applications are as follows, according to the Jakarta EE 11 Release Plan (https://eclipse-ee4j.github.io/jakartaee-platform/jakartaee10/JakartaEE10ReleasePlan):

- Jakarta EE Platform 10 (updated feature)
- Jakarta EE Web Profile 10 (updated feature)
- Jakarta Activation 2.1 (updated feature)
- Jakarta Annotations 2.1 (updated feature for Web Profile)
- Jakarta Authentication 3.0 (updated feature for Web Profile)
- Jakarta Authorization 2.1 (updated feature)
- Jakarta Batch 2.1 (updated feature)
- Jakarta Bean Validation 3.0 (feature for Web Profile)
- Jakarta Concurrency 3.0 (updated feature for Web Profile)
- Jakarta Connectors 2.1 (updated feature)
- Jakarta Contexts and Dependency Injection 4.0 (updated feature)
- Jakarta Debugging Support for Other Languages 2.0 (feature for Web Profile)
- Jakarta Dependency Injection 2.0 (feature for Web Profile)
- Jakarta Enterprise Beans 4.0
- Jakarta Enterprise Beans 4.0 Lite (feature Web Profile)
- Jakarta Enterprise Web Services 2.0 (optional feature)
- Jakarta Expression Language 5.0 (updated feature for Web Profile)
- Jakarta Interceptors 2.0 (feature for Web Profile)

- Jakarta JSON Binding 3.0 (updated feature for Web Profile)
- Jakarta JSON Processing 2.1 (updated feature Web Profile)
- Jakarta Mail 2.1 (updated feature)
- Jakarta Managed Beans 2.0 (Web Profile)
- Jakarta Messaging 3.1 (updated feature)
- Jakarta Persistence 3.1 (updated feature for Web Profile)
- Jakarta RESTful Web Services 3.1 (updated feature for Web Profile)
- Jakarta Security 3.0 (updated feature for Web Profile)
- Jakarta Server Faces 4.0 (updated feature for Web Profile)
- Jakarta Server Pages 3.1 (updated feature for Web Profile)
- Jakarta Servlet 6.0 (updated feature for Web Profile)
- Jakarta SOAP with Attachments 3.0 (optional feature)
- Jakarta Standard Tag Library 3.0 (feature for Web Profile)
- Jakarta Transactions 2.0 (feature for Web Profile)
- Jakarta WebSocket 2.1 (feature for Web Profile)
- Jakarta XML Binding 4.0 (optional feature)
- Jakarta XML Web Services 4.0 (optional feature)

At the time of this book's writing, the tool that contains portable modules for low-level authorization had not yet been incorporated into Jakarta Security 1.0, as it has not yet been fully evaluated. For example, to have a solution that identifies the scope of an application, checking for the bridging role against an external service is the best strategy, instead of taking all the roles and assigning them when the invocation call is authenticated. Such an example is shown here:

```
1   import java.io.UnsupportedEncodingException;
2   import java.security.KeyPair;
3   import java.security.KeyPairGenerator;
4   import java.security.NoSuchAlgorithmException;
5   import java.security.NoSuchProviderException;
6   import java.security.PrivateKey;
```

```
import java.security.PublicKey;
import java.security.SecureRandom;

@ApplicationScoped
public class ApressAuthorizationModule {

    @Inject
    SecurityConstraints apressSecurityConstraints

    @Inject
    ApressService apressService;

    @PostAuthenticate
    @PreAuthorize
    @ByRole
    public Boolean appressLogic(
        Caller apressCaller, Permission apressReqPermission) {

        return apressSecurityConstraints.getRequiredRoles(apressReq
        Permission)
                .stream()
                .anyMatch(apressAccessRole -> apressService.
                isInRole(apressCaller, apressAccessRole));
    }

}
```

Jakarta CDI represents the main component model within the platform, and many components are being aligned with it already. The important components include the following:

- The `@Transactional` annotation gets a new definition in Jakarta Transactions.
- The managed bean annotations is being deprecated in JSF.
- CDI has been developed and improved in Jakarta Security.

The following components related to Jakarta EE 11 take full advantage of CDI:

- When CDI is enabled, the features of the Jakarta Enterprise Beans programming model will be more flexible. Because of the modernization process within Jakarta EE 11, the features are converted to specifications where they fit perfectly. Here are some examples:
  - Definitions for `@Lock`, `@Schedule`, `@Asynchronous`, and `@MaxConcurrency` are included. Security mechanisms have also been improved, and special mechanisms are dedicated to working with these annotations.
  - Another important definition that is included is related to the message-driven bean-style `@MessageListener` within Jakarta Messaging.
  - Jakarta Security will have two annotations that will need to deal with it: `@RolesAllowed` and `@RunAs`.
- Most of the additional mechanisms are developed in such a way that they can be injected with the help of CDI to provide deprecations of their own features with respect for CDI and easy integration within CDI. These mechanisms can be summarized as follows:
  - Jakarta RESTful Web Services
  - Jakarta Batch
  - Jakarta Concurrency

Other new potential specifications within Jakarta EE 11 are the following:

- Jakarta NoSQL and Eclipse JNoSQL for giving access to Jakarta EE applications
- Jakarta Model-View-Controller (MVC)

Next, looking at the future of Jakarta EE 11, there are new challenges, but they will not be covered. Most of the components provide experimentation together with analysis and development. Those features are summarized as follows:

- Jakarta Servlet or a variant much lighter
- Jakarta Persistence, with support for reactive NIO Java Database Connectivity (JDBC)
- Jakarta Restful Web Services
- Jakarta MVC

The Java platform and Jakarta EE 11 provide a number of important built-in providers for implementing basic and advanced security services that are used within different types of projects. The main components that work in both Java Platform and Jakarta EE 11 can be summarized as follows:

- **Java Cryptographic Extension (JCE).** JCE represents an extension from Sun Microsystems for providing encryption and decryption operation for data blocks. JCE represents an important component of the Java Cryptography Architecture (JCA) implementation.
- **Java Secure Socket Extension (JSSE).** One of the most important data protocols is represented by the Secure Sockets Layer (SSL), which provides data integrity by providing encryption. JSSE has a standard interface and proper implementation for the SSL protocol.
- **Java Authentication and Authorization Service (JAAS).** JAAS offers an implementation for user authentication. It provides a wide range of login mechanisms within a specific architecture and a dedicated API.

Good references specialized for Jakarta EE are [1–3] and generally for Java are [4–7].

Conclusion

In this chapter, we provided a short overview of the main components and challenges covered by the new Jakarta EE 11 and how these components will impact the developing process of the software applications and the implementation of security mechanisms. An important aspect at this stage of development is represented by the new security authentication mechanisms that have been released. You should now be able to describe and understand the main challenges and components within Jakarta EE 11.

References

[1] Juneau, Josh. *Jakarta EE Recipes: A Problem-Solution Approach, Apress*, 2020.

[2] Manelli, Luciano, and Giulio Zambon. *Beginning Jakarta EE Web Development: Using JSP, JSF, MySQL, and Apache Tomcat for Building Java Web Applications*. Apress, 2020.

[3] Saeed, Luqman. *Introducing Jakarta EE CDI Contexts and Dependency Injection for Enterprise Java Development*. Apress : Imprint: Apress, 2020.

[4] Horstmann, Cay S. *Core Java*. Eleventh edition, Pearson, 2019.

[5] Bloch, Joshua. *Effective Java*. Third edition, Addison-Wesley, 2018.

[6] McLaughlin, Brett, et al. *Head First Object-Oriented Analysis and Design*. 1st ed, O'Reilly, 2007.

[7] Sierra, Kathy, and Bert Bates. *Head First Java*. 2nd ed, O'Reilly, 2005.

CHAPTER 4

Java Cryptography Architecture

The Java Cryptography Architecture (JCA) framework offers capabilities for working with cryptography algorithms and offers methods for a variety of cryptography primitives. JCA was introduced as part of the Java Development Kit (JDK) and offers a framework and implementation for encryption, key generation and agreement, and Message Authentication Code (MAC) algorithms. The cryptography algorithms and methods are implemented and offered through the `java.security` package.

The Java platform is designed in such a way that the most important aspects and components that characterize security are included in an easy and modern approach for developers. The following are the most essential components:

- **Secure communication.** JCA provides a robust and secure platform for implementing cryptography in Java-based applications. It simplifies complex cryptographic operations with easy-to-use APIs, minimizing the risk of errors that may lead to security breaches. Additionally, it is regularly reviewed and updated to address new security threats and vulnerabilities.
- **Cryptographic functions.** JCA includes different cryptographic functionalities, such as digital signatures, message digests (hashes), certificates and certificate validation, encryption (both symmetric and asymmetric), key generation, and generation of secure random numbers.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_4

- **Authentication.** JCA facilitates the implementation of authentication mechanisms in Java applications by offering an extensive collection of APIs and services that guarantee the validity of user identities. It allows users to create and verify digital signatures using cryptographic keys, providing authenticity and provenance verification for documents and messages. MAC algorithms also enable authentication and ensure the integrity of messages through the exchange of a mutually known secret between the sender and the receiver. Also, they establish integration with Java's KeyStore, an important component for public key infrastructure (PKI) systems that rely on certificates authenticated by trusted certificate authorities (CAs) to manage cryptographic keys and certificates securely. Furthermore, while the Java Secure Socket Extension (JSSE) is used to implement SSL and TLS, it is JCA that provides the essential cryptographic operations required to ensure the security of Internet communications, including HTTPS.
- **Access control.** Although JCA does not directly implement access control policies, its cryptographic tools and capabilities provide safe and controlled access to applications and data. Access management can be made through mechanisms like the Security Manager and Access Controller. Also, Java KeyStore provides secure key storage and certificate-based authentication and supports SSL/TLS for secure identity verification and data exchange.

JCA represents only a small part of the entire Java platform, which has a provider and a set of APIs for different cryptography primitives, such as digital signatures, hash functions, validation of certificates, encryption algorithms/primitives, cryptography key generation functions, key management, and algorithms for generating random numbers.

The provider was designed as architecture with respect for the following principles:

- **Extensibility of the algorithms.** Java is quite flexible when it comes to providing its proper cryptography algorithms or custom providers' and third-party libraries/frameworks. An important aspect that needs to be taken into consideration when developing applications and using third-party frameworks and libraries is to make sure that the

implementation is tested and standardized accordingly. Everything that is outside of the standardization process requires more attention as a developer.

- **Assuring the interoperability and implementing proper support.** As a programming language and developing technology for software applications, Java guarantees interoperability, which is an important criterion to satisfy even the most drastic requirements, especially when you are dealing with different platforms (e.g., Windows ⟷ Ubuntu ⟷ Mac) for the same applications.
- **Guaranteeing the independence of the cryptography algorithms implementation.** When you are dealing with the implementation process for security algorithms, there is no need to implement the algorithms from scratch. Thus, developers and software engineers can use the security libraries and make proper requests for the cryptography algorithms/primitives that they want to implement. The algorithms are already implemented and standardized accordingly.

JCA when used with JDK has two main components.

- The first component introduces the definitions and provides support for different cryptographic services. Different contributors provide different versions of implementations. The framework provides the following important packages:
  - `java.security.` This is Java's security framework core package that provides key generation, security policy management, cryptographic hash functions, digital signatures, and certificate management. It implements classes and interfaces for access control, safe random number generation, and cryptographic keys and certificates to build secure Java applications used for scenarios like authentication, authorization, and secure communications.
  - `javax.crypto.` The Java Cryptography Extension (JCE) is a package that provides encryption, decryption, and secure key exchange functionality to the Java security API. This package is useful for developers who need to encrypt sensitive data using a variety of cryptographic cipher algorithms,

key generation, and agreement protocols. The JCE software supports both symmetric (e.g., AES) and asymmetric (e.g., RSA) encryption, allowing for secure encryption and decryption of data.

  - `javax.crypto.spec.` This package contains classes that are specifically used to instantiate keys and parameters for cryptographic operations in the `javax.crypto` package. For instance, it defines classes like `IvParameterSpec` for specifying initialization vectors for ciphers in block mode or `SecretKeySpec` for defining keys for symmetric encryption algorithms.
  - `javax.crypto.interfaces.` This package provides interfaces for key objects specific to particular cryptographic algorithms. It includes key interfaces for algorithms like the Diffie-Hellman key exchange. For example, it defines the `DHPublicKey` and `DHPrivateKey` interfaces, which represent public and private keys in the context of the Diffie-Hellman key exchange.

- The second component, `javax.crypto`, provides the actual implementation of the cryptography algorithms/primitives. The providers are as follows:
  - `Sun`
  - `SunRsaSign`
  - `SunJCE`

Architecture and Design Principles

As we mentioned, assuring independent implementation and interoperability, along with the independence and extensibility of algorithms, plays an important role and were the main principles considered when JCA was designed and implemented.

As an example, you can use a digital signature or hash function without considering how the algorithms were implemented. So, you don't have to worry about the details of the implementation or the algorithms that represent the foundation for the concepts.

One of the fundamental questions that comes up is, how is the independence of the algorithm fulfilled? The main thing that a developer has to do is to define the cryptography engines or services along with the definition of the classes that offer

capabilities and different functionalities specific to those cryptographic engines. The name of the concepts, according to some reference guides [1], is known as *engine classes*. There are a few examples that help in this area, shown here:

- `Cipher` class
- `KeyPairGenerator` class
- `KeyFactory` class
- `MessageDigest` class
- `Signature` class

Another question that will come up is, how is independence implemented? The cryptographic service provider (CSP) represents a package or a set of packages that provides implementations for a variety of cryptographic services, such as digital signatures algorithms, hash algorithms, and key management algorithms. Simply stated, interoperability in information security is the ability of different implementations to achieve their goals by working with each other, exchanging cryptography keys between them, or being able to check each other's cryptography keys.

The final question is, what is the importance and meaning of algorithm extensibility? The answer relies in the simplicity of how the engine classes are added in a very easy and modern way.

The base class `java.security.Provider` comprises all the security providers. For each CSP we will have an instance of this class. The class will offer the following:

- A list with all the security algorithms and security services that are implemented
- The provider name

To use JCA, you simply specify the request for a certain object and its type. With every JDK that you install, one or more providers will be installed and set by default.

Let's see an example of JCA in action. The first thing you need to do is to make a simple request for a certain type of object, such as `MessageDigest`, and to invoke or call a certain algorithm/service, such as the SHA-512 algorithm.

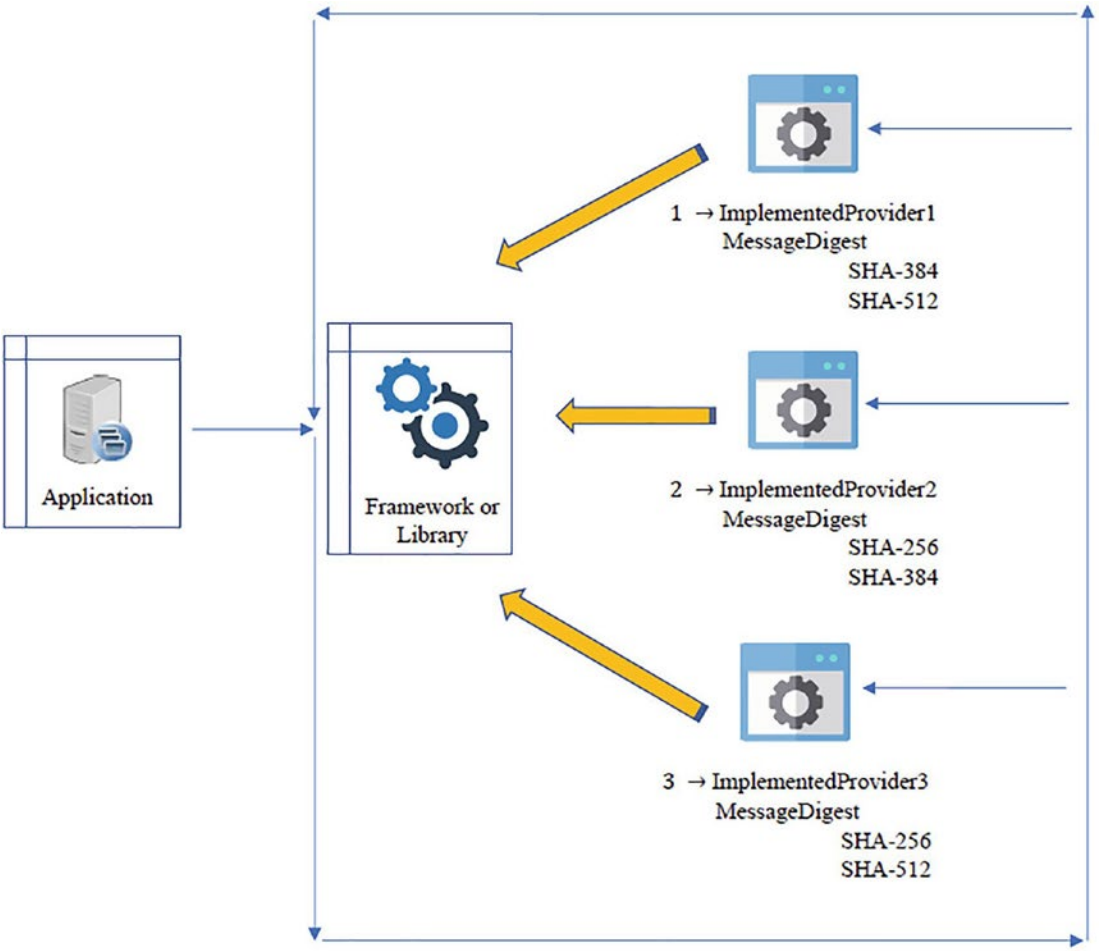
```
message_hash = MessageDigest.getInstance("SHA-512");
message_hash = MessageDigest.getInstance("SHA-384",
"ImplementedProvider1");
```

Note that the previous lines of code are just an example snippet. In real use, `ImplementedProvider1` should be replaced with a valid provider. For example, a provider for SHA-384 is `SUN`. Java enables users to determine the providers available by using the following code:

```
Provider[] providers = Security.getProviders();
```

Figure 4-1 and Figure 4-2 show the requesting process for the SHA-512 algorithm to hash a message. The ordering process shows three different ways of how a hash algorithm is invoked and how a provider can be used as a source of the implementation. Based on the architectures shown in Figure 4-1 and Figure 4-2, the developers can add any other components (e.g., services, interfaces, third-party functions, etc.) and develop new architectures based on them.

Returning to the independence of the algorithms, their definition will achieve the proper goal of providing the implementation of providers that are compliant with the defined interfaces. This is known as an *application programming interface* (API), and it will guarantee that all applications will use it to access a specific type of service.

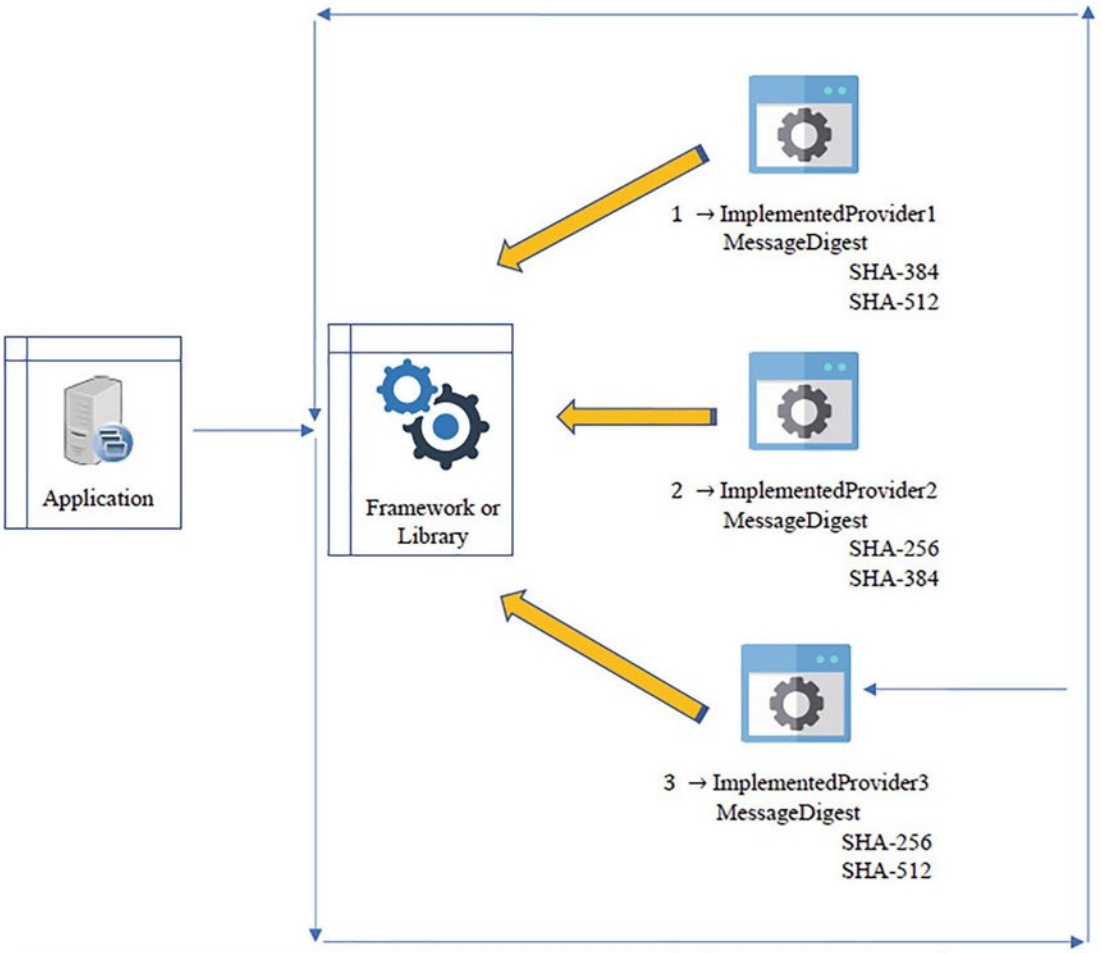


Legend:

- → Shows the path and the way how the provider is acting. Precisely, it represents the call of the *getInstance* method from *MessageDigest* class and how the call starts from the Application components and goes through the next components indicated by the arrow, such as:
  1. Framework or Library
  2. ImplementedProvider1
  3. ImplementedProvider2
  4. Framework or Library – it comes back.

More details about the scenario can be seen at:
https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

Figure 4-1. *Searching process for provider*



Legend:

- ⟶ Shows the path and the way how the provider is acting. Precisely, it represents the call of the *getInstance* method from *MessageDigest* class and how the call starts from the Application components and goes through the next components indicated by the arrow, such as:

1. Framework or Library
2. ImplementedProvider3
3. Framework or Library – it comes back.

The *getInstance* will return the SHA-256 implementation for hashing the message from the *ImplementedProvider3*.

More details about the scenario can be seen at:
https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

Figure 4-2. *Requesting a specific provider*

JCA Classes and Algorithms

This section presents the classes and algorithms within JCA.

Algorithms and Engine Classes

The *engine class* refers to a type of framework or abstract class that defines a cryptographic service in a generic way. Engine classes are essentially templates for implementing different cryptographic operations such as encryption, decryption, message digest calculation, signing, and key generation. They provide a standardized interface through which these operations can be accessed and performed, regardless of the underlying algorithm or provider implementation. An engine class offers the following operations and components:

- **Objects.** The objects contain the encrypted data and have the capability to move the data to certain and upper abstraction layers. Such examples include certificates or cryptography key containers.
- **Cryptography primitive operations.** These include encryption/decryption operations, digital signatures, hashing messages, etc.
- **Generators or convertors.** These include mechanisms used for different cryptography concepts/mechanisms, such as cryptography keys or various parameters for cryptography algorithms.

You can find a comprehensive list with examples and case studies at [1]. Table 4-1 lists the most important ones.

Table 4-1. *Engine Classes and Their Purpose*

Engine Class	Purpose
`KeyPairGenerator`	This generates a pair of public and private keys for a certain algorithm.
`KeyGenerator`	For an algorithm, the class will generate new secret keys.
`KeyAgreement`	The instance of the class is used by one or two parties to establish a cryptographic key to be used within a specific cryptographic operation.

(*continued*)

Table 4-1. (*continued*)

Engine Class	Purpose
`Cipher`	The class will use the keys (generated for encryption and decryption data). The class provides a set of various algorithms, such as asymmetric encryption, symmetric encryption, and password-based encryption.
`MessageDigest`	This provides functionalities for computing the hash value for a certain message.
`SecureRandom`	This provides functionalities for generating random numbers.

Interfaces and Main Classes

Table 4-2 lists the main classes and interfaces that JCA deals with. The list has been designed based on how useful and frequently used a class/interface is.

Table 4-2. *Classes/Interfaces and Their Purpose*

Class/Interface	Purpose
`Provider`	This generates a pair of public and private keys for a certain algorithm.
`Security`	For an algorithm, this class will generate new secret keys.
`MessageDigest`	An instance of this class is used by one or two parties for establishing a cryptographic key to be used within a specific cryptographic operation.
`Cipher`	This class will use the keys (generated for encryption and decryption data). The class provides a set of various algorithms, such as asymmetric encryption, symmetric encryption, and password-based encryption.
`KeyAgreement`	This provides functionalities for computing the hash value for a certain message.
`CertificateFactory`	This provides functionalities for generating random numbers.
`Key`	This provides functionalities for working with keys.

For a comprehensive list, the resource listed in [1] should be consulted.

The most important packages that are used in practice are `java.security`, `javax.crypto`, `javax.crypto.interfaces`, `java.security.interfaces`, `javax.crypto.spec`, `java.security.cert`, and `java.security.spec`.

Data Encryption

Data encryption is essential to protect information from unauthorized access and ensure its confidentiality. It involves transforming plain data, called *plaintext*, into an unreadable format, called *ciphertext*, using cryptographic algorithms based on encryption keys. This process ensures that even if malicious actors intercept or access the data, it cannot be decrypted without the corresponding decryption key. Modern encryption algorithms, such as Advanced Encryption Standard (AES) [3] and Rivest-Shamir-Adleman (RSA) [4], offer strong security by making it extremely difficult (computationally infeasible) to decrypt the encrypted data without the correct key.

To work with data encryption, the `Cipher` class offers the right tools (e.g., functions, methods, etc.). The `Cipher` class is part of the `javax.crypto` package.

The workflow is straightforward to follow, as shown here and in Listing 4-1:

1. Declare a `KeyPairGenerator` object.
2. The `KeyPairGenerator` has to be initialized accordingly, specifying the algorithm that will be used.
3. Using the method `generateKeyPair()`, you will generate the key pair.
4. Obtain the public key using `getPublic()`.
5. Create an instance for the cipher object. To achieve this goal, the method `getInstance()` is used.
6. Using the `init()` method, you will initialize the cipher object.
7. Provide the data to the cipher object for which you will provide the encryption.
8. Provide the encryption of the data.

Listing 4-1. Encryption Example

```
1  import java.security.KeyPair;
2  import java.security.KeyPairGenerator;
3  import java.security.Signature;
4
5  import javax.crypto.Cipher;
```

```
 6
 7  public class DataEncryption {
 8     public static void main(String args[]) throws Exception{
 9
10        //create an instance for signature
11        Signature mySignature = Signature.getInstance("SHA256withRSA");
12        System.out.println("\n The signature instance -> " +
          mySignature);
13
14        //create an instance as generator for a pair key
15        KeyPairGenerator key_pair_gen = KeyPairGenerator.
          getInstance("RSA");
16        System.out.println("\n The key pair generator -> " + key_
          pair_gen);
17
18        //specify the size
19        key_pair_gen.initialize(2048);
20        System.out.println("\n The key pair generator size -> " + key_
          pair_gen);
21
22        //create the key pair instance
23        KeyPair pair = key_pair_gen.generateKeyPair();
24        System.out.println("\n The instance of the key pair -> "
          + pair);
25
26        //specify the algorithm as instance
27        Cipher algorithm = Cipher.getInstance("RSA/ECB/PKCS1Padding");
28        System.out.println("\n The algorithm -> " + algorithm);
29
30        //choose the mode (encryption/decryption)
31        algorithm.init(Cipher.ENCRYPT_MODE, pair.getPublic());
32        System.out.println("\n The algorithm with the mode -> " +
          algorithm);
33
34        //choose a text that will be encrypted
```

```
35          byte[] input = "Learning Cryptography with Java".getBytes();
36          algorithm.update(input);
37          System.out.println("\n The algorithm with the mode -> " +
            algorithm);
38
39          //encrypting the data
40          byte[] encrypted_text = algorithm.doFinal();
41          System.out.println("\n Encrypted text -> " + new
            String(encrypted_text, "UTF8"));
42      }
43  }
```

Figure 4-3 shows the output.

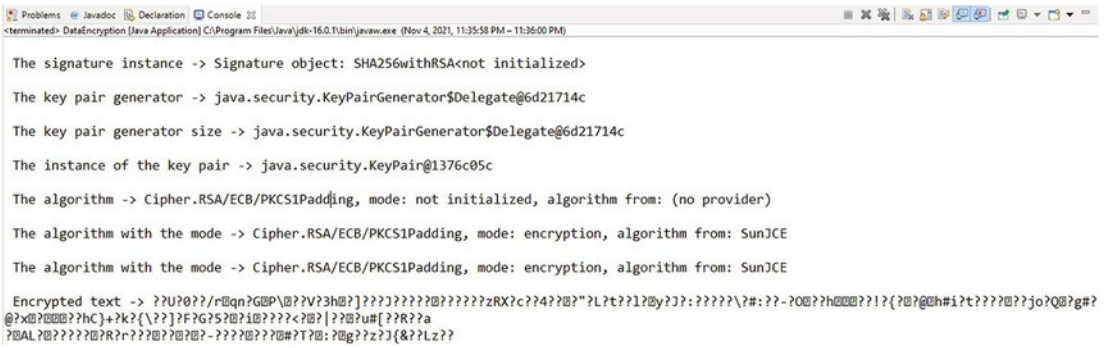

Problems Javadoc Declaration Console
<terminated> DataEncryption [Java Application] C:\Program Files\Java\jdk-16.0.1\bin\javaw.exe (Nov 4, 2021, 11:35:58 PM – 11:36:00 PM)
The signature instance -> Signature object: SHA256withRSA<not initialized>
The key pair generator -> java.security.KeyPairGenerator$Delegate@6d21714c
The key pair generator size -> java.security.KeyPairGenerator$Delegate@6d21714c
The instance of the key pair -> java.security.KeyPair@1376c05c
The algorithm -> Cipher.RSA/ECB/PKCS1Padding, mode: not initialized, algorithm from: (no provider)
The algorithm with the mode -> Cipher.RSA/ECB/PKCS1Padding, mode: encryption, algorithm from: SunJCE
The algorithm with the mode -> Cipher.RSA/ECB/PKCS1Padding, mode: encryption, algorithm from: SunJCE
Encrypted text ->

Figure 4-3. *The output of the encryption process*

Hash Functions

Hash functions generate a fixed-size, unique "digest" or "hash" value from input data of any size [5, 6]. These functions are designed to be one-way, meaning it is computationally infeasible to reverse the process and retrieve the original input from the hash output. Because of this characteristic, hash functions are useful for various security applications, such as verifying data integrity, authenticating messages, and securely storing passwords. When data is hashed, any change to the original input, even a minor shift in a bit, results in a drastically different hash. Thus, any change is detected from the original input. Examples of hash functions include SHA-256 and MD5, although MD5's vulnerabilities have led to a preference for more secure alternatives like SHA-256 and SHA-3.

To use hash functions (Listing 4-2 and Figure 4-4), the process is simple and generally is composed of three steps, as follows:

1. Provide an instance for the `MessageDigest` class.
2. Initialize the `MessageDigest` instance with data.
3. Generate the hash of the provided message.

Listing 4-2. Generating Hash Functions

```
import java.security.MessageDigest;
import java.util.Scanner;

public class Hashing {
   public static void main(String args[]) throws Exception
   {
      //add some text that we will use it for encryption
      String message = "Welcome to Appress and enjoy learning
      cryptography";
      System.out.println("\n The text for which we will provide the
      digest (hash) is -> " + message);

      //declare an instance for which we will provide the hash digest
      using SHA-256
      MessageDigest messsage_digest = MessageDigest.
      getInstance("SHA-256");

      //get the bytes representation by providing data for updating
      the digest
      messsage_digest.update(message.getBytes());

      //obtain the hash of the message
      byte[] digest = messsage_digest.digest();
      System.out.println(digest);

      //provide conversion from byte to hex string format
      StringBuffer hex_representation = new StringBuffer();
```

```
23
24          for (int i = 0;i<digest.length;i++)
25          {
26               hex_representation.append(Integer.toHexString(0xFF &
                 digest[i]));
27          }
28          System.out.println("The hex representation : " + hex_
            representation.toString());
29      }
30  }
```

The text for which we will provide the digest (hash) is -> Welcome to Appress and enjoy learning cryptography
[B@41cf53f9
The hex representation : b2601ff0a0ff94bbd0d5f398b5412ccbc471592b6a20be7a8f38893ae6f1144e

Figure 4-4. *The hash function output*

Signatures

Creating and working with signatures in Java represents an interesting approach compared with other programming languages. Digital signatures give you the ability to verify different aspects to make certain a message is genuine, such as the following [7, 8]:

- Verifying the author
- Checking the date and time of signatures
- Providing authentication of the message contents

Digital signatures offer the following advantages:

- **Authentication:** Digital signatures will help you properly authenticate the source that is sending the message(s).
- **Integrity:** The signature will be invalided if any change is made on the message.
- **Nonrepudiation:** Once a signature has been provided by someone or an entity, there is no chance at a certain point to deny the process of signing.

Generating Signatures

The implementation process of generating signatures involves eight steps, as follows:

1. Declare an instance of the `KeyPairGenerator` class.
2. Provide the proper initialization for the instance.
3. Obtain the key pair using the `generateKeyPair()` method.
4. Extract the private key from the pair using the `getPrivate()` method.
5. Declare an instance as a signature object.
6. Initialize the signature object.
7. Provide the data for the signature object.
8. Compute the signature.

Listing 4-3 shows how the steps listed are implemented, with an explanation of each step. Figure 4-5 shows the result.

Listing 4-3. Example for Generating a Signature

```
1  import java.security.*;
2  import java.util.Scanner;
3
4  public class GeneratingSignature
5  {
6     public static void main(String args[]) throws Exception {
7        //the message that will be signed
8        String data_message = "Welcome to Apress and enjoy learning
         cryptography";
9        System.out.println("The message that will be sign is -> " +
         data_message);
10
11       //Step 1 - Declare an instance of KeyPairGenerator class
12       KeyPairGenerator generatorKeyPair = KeyPairGenerator.
         getInstance("DSA");
13
```

```
14          //Step 2 - Providing the proper initialization for the instance
15          generatorKeyPair.initialize(2048);
16
17          //Step 3 - Obtaining the key pair using the
            generateKeyPair() method
18          KeyPair cryptographyKeyPair = generatorKeyPair.
            generateKeyPair();
19
20          //Step 4. Extract the private key from the pair using the
            getPrivate() method
21          PrivateKey private_key = cryptographyKeyPair.getPrivate();
22
23          //Step 5. Declare an instance as a signature object
24          Signature signatureForMessage = Signature.
            getInstance("SHA256withDSA");
25
26          //Step 6. Initialize the signature object
27          signatureForMessage.initSign(private_key);
28          byte[] theBytesRepresentation = "msg".getBytes();
29
30          //Step 7. Provide the data for the signature object
31          signatureForMessage.update(theBytesRepresentation);
32
33          //Step 8. Compute the signature
34          byte[] signature = signatureForMessage.sign();
35
36          //Showing the result as output
37          System.out.println("Obtained digital signature for the provided
            text ->  "+new String(signature, "UTF8"));
38      }
39  }
```

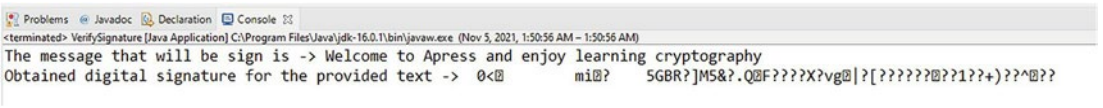


Figure 4-5. *Output for generating a signature*

Verifying the Signature

The verifying signature process is different from the generating process in terms of the steps that you need to follow for the implementation. The steps are as shown here:

1. Declare an instance of the `KeyPairGenerator` class.
2. Provide the proper initialization for the instance.
3. Obtain the key pair using the `generateKeyPair()` method.
4. Extract the private key from the pair.
5. Declare an instance for the signature object.
6. Provide the proper initialization for the signature object.
7. Provide data for the signature instance.
8. Compute the signature.
9. Provide verification for the signature instance.
10. Provide verification for the data by updating the signature.
11. Invoke the verifying process for the signature.

Listing 4-4 shows how the steps are implemented, and it points out the main differences between the generating process and the verifying process. Figure 4-6 shows the results.

Listing 4-4. Implementation for Generating and Validating a Signature

```
1  import java.security.*;
2  import java.util.Scanner;
3
4  public class VerifyingSignature
5  {
6     public static void main(String args[]) throws Exception
```

```
7       {
8           //The message used for generating and verifying process of
            the signature
9           String message_data = "Welcome to Apress and enjoy learning
            cryptography";
10          System.out.println("The message used -> " + message_data);
11
12          //Step 1. Declare an instance of KeyPairGenerator class
13          KeyPairGenerator generating_key_pair = KeyPairGenerator.
            getInstance("DSA");
14
15          //Step 2. Providing the proper initialization for the
            instance
16          generating_key_pair.initialize(2048);
17
18          //Step 3. Obtaining the key pair using the
            generateKeyPair() method
19          KeyPair cryptographic_pair = generating_key_pair.
            generateKeyPair();
20
21          //Step 4. Extract the private key from the pair
22          PrivateKey private_crypto_key = cryptographic_pair.
            getPrivate();
23
24          //Step 5 - Declare an instance for the signature object
25          Signature signature = Signature.getInstance("SHA256withDSA");
26
27          //Step 6. Providing the proper initialization for the
            signature object
28          signature.initSign(private_crypto_key);
29          byte[] bytes = message_data.getBytes();
30          System.out.println("The representation as bytes is: "
            + bytes);
31
32          //Step 7. Provide data for the signature instance
```

```
33            signature.update(bytes);
34
35            //Step 8. Compute the signature
36            byte[] computedSignature = signature.sign();
37
38            //Step 9. Provide verification for the signature instance
39            signature.initVerify(cryptographic_pair.getPublic());
40            signature.update(bytes);
41
42            //Step 10. Provide verification for the data through updating
              the signature, and
43            //Step 11. Invoke the verifying process for the signature
44            boolean bool = signature.verify(computedSignature);
45
46            if(bool)
47            {
48                  System.out.println("The signature has been checked
                    (verified) with success");
49            }
50             else
51            {
52                  System.out.println("The signature has been failed");
53             }
54        }
55  }
```

```
Problems  Javadoc  Declaration  Console
<terminated> VerifyingSignature [Java Application] C:\Program Files\Java\jdk-16.0.1\bin\javaw.exe (Nov 5, 2021, 2:28:46 AM – 2:28:48 AM
The message used -> Welcome to Apress and enjoy learning cryptography
The representation as bytes is: [B@b1a58a3
The signature has been checked (verified) with success
```

Figure 4-6. *The verifying process*

Conclusion

This chapter covered some of the main classes specific to the Java Cryptography Architecture. The chapter explained the basic concepts of JCA; understanding how the JCA architecture was designed plays an important role in developing professional software applications.

You should now understand the basic process for implementing regular cryptography operations using JCA, including the following:

- Data encryption
- Hash functions
- Digital signatures (generating and verifying process)

References

[1] Java Cryptography Architecture (JCA) Reference Guide, https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#Design.

[2] Java Downloads, https://www.oracle.com/java/technologies/downloads/#java17.

[3] J. Daemen and V. Rijmen, 'AES proposal: Rijndael', 1999, Accessed: Mar. 30, 2024. [Online]. Available: https://www.cs.miami.edu/home/burt/learning/Csc688.012/rijndael/rijndael_doc_V2.pdf

[4] R. L. Rivest, A. Shamir, and L. Adleman, 'A method for obtaining digital signatures and public-key cryptosystems', Commun. ACM, vol. 21, no. 2, pp. 120–126, Feb. 1978, doi: 10.1145/359340.359342.

[5] A. Mittelbach and M. Fischlin, The Theory of Hash Functions and Random Oracles: An Approach to Modern Cryptography. in Information Security and Cryptography. Cham: Springer International Publishing, 2021.

[6] B. Preneel, 'Cryptographic hash functions', Trans Emerging Tel Tech, vol. 5, no. 4, pp. 431–448, Jul. 1994.

[7] J. Chia, J.-J. Chin, and S.-C. Yip, 'Digital signature schemes with strong existential unforgeability', F1000Research, vol. 10, 2021, [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9925878/.

[8] V. Srivastava, A. Baksi, and S. K. Debnath, 'An overview of hash based signatures', Cryptology ePrint Archive, 2023. [Online]. Available: https://eprint.iacr.org/2023/411.

CHAPTER 5

Classical Cryptography

Humans have been hiding messages from the very first moment they started sending messages. In the early ages, messages were sent between leaders, emperors, conquerors, kings, and so on, in the form of verbal messages through a human messenger. Although unspoken rules protected the human messengers, sometimes this rule was not followed, and the messenger was caught by enemies. At that point, all the enemy needed to do was send an altered message to the rightful receiver. This is a real-life example of a *man-in-the-middle* attack.

In the years that followed, humans started hiding the actual content of the message in specific ways so that if the enemy took possession of the message, its meaning could not be understood. In this way, an early form of cryptography was born.

Methods of encrypting messages are as limitless as the human imagination. However, in practice, only some of them work well due to some natural constraints. For example, the sender and the receiver should both know how to encrypt and decrypt the messages, and the encryption technique should be strong enough such that if the encrypted message gets into the wrong hands, it cannot be decrypted. These two simple facts haunt all encryption schemes even nowadays. In this chapter, some historical ciphers will be presented, as follows: Caesar cipher, Vigenère cipher, and Hill cipher.

All these classical ciphers fall into the category of *symmetric cryptography*, in which just one type of key is used for both encryption and decryption, namely, the *secret key* (we'll discuss symmetric cryptography more in Chapter 10). Although all these classical ciphers are not used anymore, they are important as an introduction to cryptography so you can understand the basic concepts and the purpose behind cryptography. Also, they are easy to understand from a technical perspective, which makes them again a great introduction to cryptography, especially for beginners in this field. The Caesar, Vigenère, and Hill ciphers are historically and educationally significant in the field of cryptography. With a simple *substitution approach*, the Caesar cipher is an important instrument for introducing the fundamental concepts of encryption and the weaknesses/vulnerabilities

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_5

of basic ciphers. The Vigenère cipher, with its use of a repeating keyword to vary the Caesar shift, introduces the concept of *polyalphabetic substitution*, offering a more complex system that resisted unbroken for centuries and illustrating the importance of key management. Lastly, the Hill cipher, a polygraphic substitution cipher, uses matrices and modular arithmetic, showing the application of linear algebra in cryptography and providing a bridge to more modern techniques that involve mathematical structures. Collectively, these ciphers mark significant milestones in the historical development of cryptography and also provide clear examples of encryption methods and their respective strengths and weaknesses used over centuries. Moreover, these ciphers lay the groundwork for understanding more advanced cryptographic algorithms used today. You can find more background information about the ciphers in this chapter in many cryptography courses and books, such as [1], [2], and [3].

Caesar Cipher

The Caesar cipher is estimated to date back to the first century BC in the Roman Empire. Its name comes from the Roman Emperor Julius Caesar, who used this cipher to encrypt military messages and strategies [1], and it is estimated to be broken in about the fifth century AD. The Caesar cipher is a substitution cipher, in which each letter of the alphabet is moved a certain number of characters to the right. For example, if the established number is 5, then *A* will become *F*, *B* will become *G*, *C* will become *H*, etc. In this cipher, the *key* is represented by the number that the letters are shifted. Note that the key should be secret, known only by the sender (to encrypt the messages) and by the receiver (to decrypt the messages). The Caesar cipher is important because it is the basis for other classical ciphers, such as Vigenère. The Caesar cipher with key $k = 13$ is also known as Rot13.

Let's consider an example. The key for the cipher is set to $k = 6$; therefore, all letters are shifted six characters (see Table 5-1).

Table 5-1. *Caesar Representation with $k = 6$*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F

Note that there are 26 letters in the alphabet, and in Table 5-1 the numbering starts with 0 and ends with 25 because it is related to $Z_{26} = \{0, 1, \ldots, 25\}$. Sometimes, in other documentation, the numbering starts with 1 and ends with 26, because it seems more natural.

Considering the plain message as "CAESAR CIPHER EXAMPLE" and the key as $k = 6$, the encrypted message will be "IGKYGX IOVNKX KDGSVRK." The encrypted message was obtained by replacing *C* with the corresponding *I*, *A* with the corresponding *G*, *E* with the corresponding *K*, and so on.

To decrypt the message, the letters from the second row of the table are shifted to the left for *k* positions. Therefore, the decryption of "IGKYGX IOVNKX KDGSVRK" (with the key $k = 6$) is "CAESAR CIPHER EXAMPLE," obtained by applying a similar logic as for encryption: *I* is replaced with *C*, *G* is replaced with *A*, and so on. Note that sometimes the space character is removed from the message; therefore, in the example, the plain text would be "CAESARCIPHEREXAMPLE," while the encrypted text would be "IGKYGXIOVNKXKDGSVRK."

From a mathematical point of view, the operations are addition for encryption and subtraction for decryption via a modulo operation. The first step is to "convert" the characters to numbers, as follows: $'A' \to 1$, $'B' \to 2$, …, $'Z' \to 26$. The encryption function enc_k will take as input one parameter, i.e., the number corresponding to the letter that needs to be encrypted at the current step, and will output the encrypted letter:

$$enc_k(c) = (c + k) \ mod \ 26,$$

where *c* is the plain character and *k* is the key. The decryption function needs to shift back the letters and, therefore, has the following representation:

$$dec_k(c') = (c' - k) \ mod \ 26,$$

where c' is the encrypted character that needs to be decrypted and *k* is the key.

Implementation

Listing 5-1 shows the implementation of the Caesar cipher; Figure 5-1 shows the result.

Listing 5-1. Implementation of the Caesar Cipher

```
1   package ciphers;
2
3   import java.util.Scanner;
4
5   public class CaesarCipher {
6
7       public static String Encrypt(String text, int positions) {
8           StringBuilder toEncrypt = new StringBuilder();
9           // Format the text to be encrypted
10          for (char c : text.toCharArray()) {
11              // Remove space characters
12              if (c != ' ') {
13                  // If the character is lowercase, make it uppercase
14                  toEncrypt.append(Character.toUpperCase(c));
15              }
16          }
17
18          StringBuilder result = new StringBuilder();
19          for (int i = 0; i < toEncrypt.length(); i++) {
20              // Shift the current letter of the message with given
                positions to the right
21              char shiftedLetter = (char) (toEncrypt.charAt(i) +
                positions);
22              // If the ASCII code exceeds 'Z', then bring it back in the
                interval A..Z
23              if (shiftedLetter > 'Z') {
24                  shiftedLetter = (char) (shiftedLetter - 26);
25              }
26              result.append(shiftedLetter);
27          }
28          return result.toString();
29      }
30
```

```
31      public static String Decrypt(String text, int positions) {
32          StringBuilder result = new StringBuilder();
33          for (char c : text.toCharArray()) {
34              // Shift the current letter of the message with given
                positions to the left
35              char shiftedLetter = (char) (c - positions);
36              // If the ASCII code goes below 'A', then bring it back in
                the interval A..Z
37              if (shiftedLetter < 'A') {
38                  shiftedLetter = (char) (shiftedLetter + 26);
39              }
40              result.append(shiftedLetter);
41          }
42          return result.toString();
43      }
44
45      public static void main(String[] args) {
46          System.out.println("CAESAR CIPHER\n");
47
48          try (Scanner sc = new Scanner(System.in)) {
49              // Reading the input: plain message and the secret key
50              System.out.print("Type the message: ");
51              String message = sc.nextLine();
52              System.out.print("Type the key: ");
53              int key = sc.nextInt();
54
55              // Encrypting the plain message
56              System.out.println("\nEncrypting...");
57              String encryptedMessage = Encrypt(message, key);
58              System.out.println("The encrypted message is: " +
                encryptedMessage);
59
60              // Decrypting the encrypted message
61              System.out.println("\nDecrypting...");
62              String recoveredMessage = Decrypt(encryptedMessage, key);
```

```
63              System.out.println("The decrypted message is: " +
                recoveredMessage);
64          }
65      }
66  }
```

The code is pretty simple and just follows the encryption and decryption formula. Before encrypting the message, it is formatted by eliminating the space characters and making all the letters uppercase.

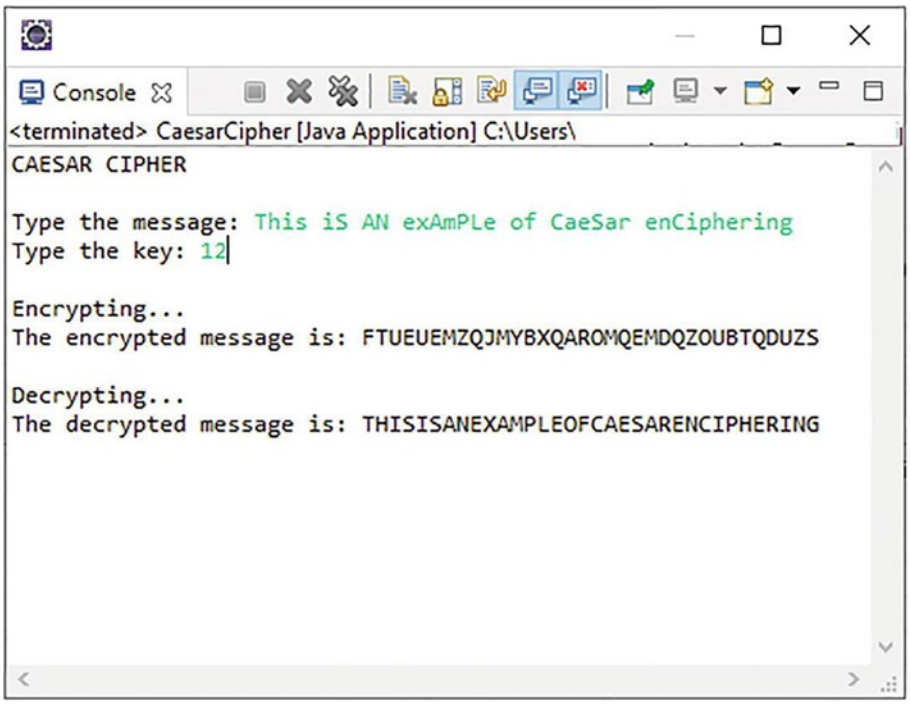


Figure 5-1. *The result of Caesar enciphering*

The following are the key features of the code from Listing 5-1:

- **Use of `StringBuilder`:** The string concatenation is made with the `StringBuilder` object to improve performance, particularly in the encryption and decryption loops.
- **Enhanced loops:** There are enhanced `for` loops (`for-each`) where appropriate for cleaner code and to avoid indexing errors.

- **Local variable type inference (`var`):** This could be used here, but since the types are clear from the declarations (`StringBuilder`, `Scanner`), using `var` would not significantly enhance readability. However, it is an option if a less verbose codding style is prefered.
- **Try-with-resources on `Scanner`:** This ensures proper closure of the resource, enhancing resource management.
- **Simplified character wrapping:** The logic to wrap characters around the alphabet using modulo arithmetic is used, making it easier to understand and reducing potential errors.

Cryptanalysis

The Caesar cipher is easy to use, but it is also easily broken in milliseconds using automated tools [5]. The total number of keys is 25; therefore, there are only 25 shifting possibilities. An attacker can easily generate all 25 forms and then recover the message for all forms. This type of attack, in which all possible values are exploited, is called a *brute-force attack*.

The brute-force attack for the Caesar cipher consists of determining the fitness value of a part of the decrypted text, showing how that part is fitting in the text. This implies some statistical information about the decrypted text compared to statistical information about English text. The fitness value shows how "close" to recover the key an attacker is, with higher values closer to recovering the key. An example of a statistical method that can be applied here is the quadgram fitness measure [4] that works as follows:

1. Compute the static values for the English text.
2. Compute the probability for the event that the ciphertext has the same distribution.

Vigenère Cipher

This historical cipher was mistakenly attributed to Blaise de Vigenère in the 18th century, but it was invented by the Italian cryptologist Giovan Battista Bellaso (`https://en.wikipedia.org/wiki/Giovan_Battista_Bellaso`), and the first use is estimated to be in the 15th century. Until 1863, when it was broken through a method proposed by Friedrich Kasiski, the Vigenère cipher was considered unbreakable, gaining the nickname "le chiffrage indéchiffrable" (in French), meaning "the indecipherable cipher."

The Vigenère cipher is a polyalphabetic substitution cipher that uses a key in the form of a word and an encryption table (see Table 5-2). To encrypt a message, it should proceed as follows:

1. Extend the key until it reaches the length of the message, by repeating circularly the letters of the key.
2. For each letter of the message, find the intersection in the encryption table between the current letter and the letter with the same index in the extended key.
3. The found letter is the encrypted letter of the current plain letter.

Table 5-2. *The Encryption Table for the Vigenère Cipher [6]*

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

To make it easier to understand, see the example in Table 5-2.

Consider the plaintext "LOREM IPSUM DOLOR SIT AMET" and the key "APRESS."

First, extend the key, as in Table 5-3. Note that the space character was removed from the plain message.

Table 5-3. *Extending the Key for Vigenère Enciphering*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Text	L	O	R	E	M	I	P	S	U	M	D	O	L	O	R	S	I	T	A	M	E	T
Key	A	P	R	E	S	S	A	P	R	E	S	S	A	P	R	E	S	S	A	P	R	E

Second, find the intersection between the current key letter and the current message letter (see Figure 5-2).

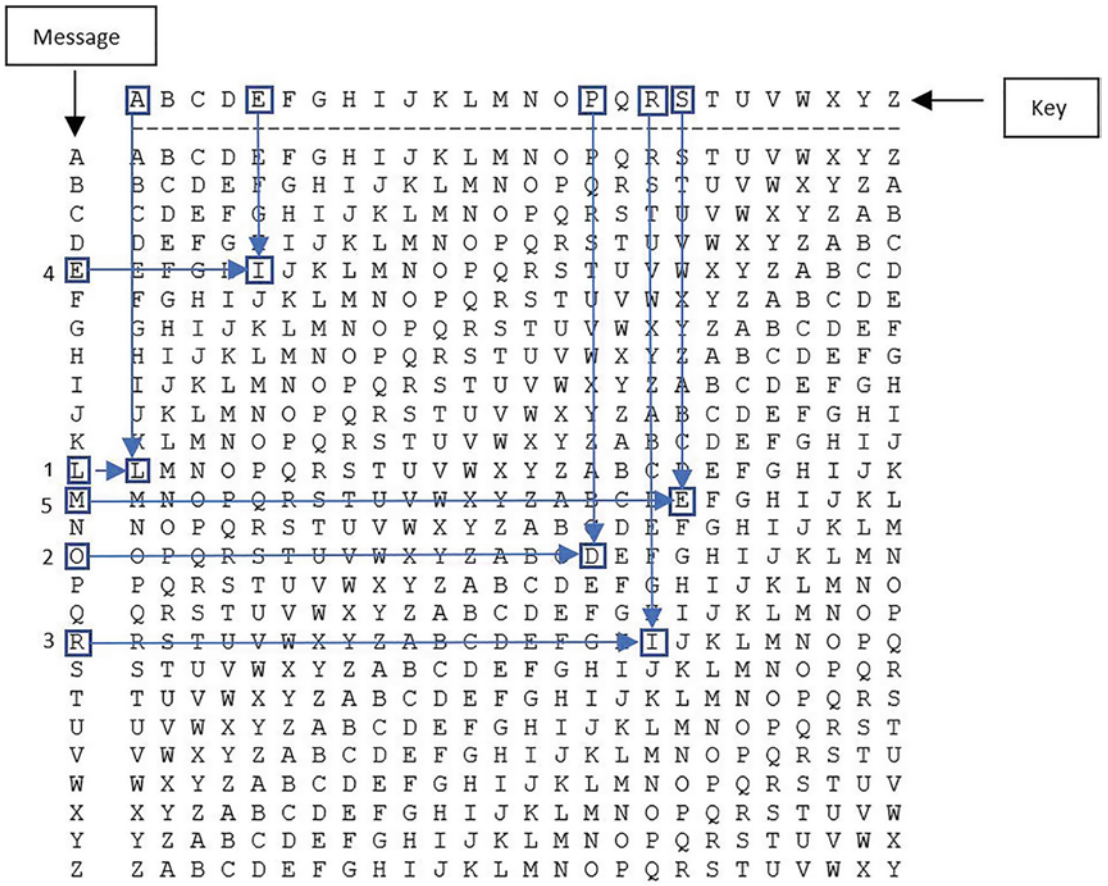


Figure 5-2. *Encryption of the first word of the message using Vigenère*

Continuing with the encryption technique, the encrypted message is "LDIIEAPHLQVGLDIWALABVX."

The Vigenère cipher is a substitution cipher in which each letter of the plain message is replaced with the letter of the intersection between the key and the plain message, as shown earlier.

The technical aspects of the Vigenère cipher are related to $Z_{26} = \{0, 1, \ldots, 25\}$. Considering m_i the current letter of the message and k_i the current letter of the extended key, the corresponding encrypted letter is computed using the following formula:

$$enc(m_i) = (m_i + k_i) \ mod \ 26$$

The decryption of the current letter c_i of the encrypted message is made using the following formula:

$$dec(c_i) = (c_i - k_i) \ mod \ 26,$$

where k_i is the current letter of the extended key.

The Vigenère cipher is important because it was the starting point of the one-time pad (OTP) proposed by Gilbert Vernam in the 1900s, which is the basis for authentication systems today.

Implementation

This section presents the implementation of the Vigenère cipher. It follows the steps described in the previous section, but first, the message is processed by eliminating the space characters and making the letters uppercase. It also uses the key features of coding presented for Listing 5-1. Listing 5-2 shows the implementation, and Figure 5-3 shows the result.

Listing 5-2. Implementation of the Vigenère Cipher

```
1   package ciphers;
2
3   import java.util.Scanner;
4
5   public class Vigenere {
```

```
6       static String ExtendKey(String message, String initialKey) {
7           StringBuilder extendedKey = new StringBuilder(initialKey);
8           int messageLength = message.length();
9           int keyLength = initialKey.length();
10
11          for (int i = 0; extendedKey.length() < messageLength; i++) {
12              extendedKey.append(initialKey.charAt(i % keyLength));
13          }
14          return extendedKey.toString();
15      }
16
17      static String Encrypt(String text, String extendedKey) {
18          StringBuilder result = new StringBuilder();
19
20          for (int i = 0; i < text.length(); i++) {
21              int charIndex = (text.charAt(i) - 'A' + extendedKey.
                charAt(i) - 'A') % 26 + 'A';
22              result.append((char) charIndex);
23          }
24          return result.toString();
25      }
26
27      static String Decrypt(String text, String extendedKey) {
28          StringBuilder result = new StringBuilder();
29
30          for (int i = 0; i < text.length(); i++) {
31              int charIndex = (text.charAt(i) - extendedKey.charAt(i) +
                26) % 26 + 'A';
32              result.append((char) charIndex);
33          }
34          return result.toString();
35      }
36
37      static String ProcessText(String text) {
38          StringBuilder result = new StringBuilder();
39
```

```
        for (char c : text.toCharArray()) {
            if (c != ' ') {
                result.append(Character.toUpperCase(c));
            }
        }

        return result.toString();
    }

    public static void main(String[] args) {
        System.out.println("VIGENERE CIPHER\n");

        try (Scanner sc = new Scanner(System.in)) {
            System.out.print("Type the message: ");
            String plainMessage = sc.nextLine();
            System.out.print("Type the key: ");
            String key = sc.nextLine();

            String processedText = ProcessText(plainMessage);
            String keyword = ProcessText(key);

            String extendedKey = ExtendKey(processedText, keyword);
            String encryptedText = Encrypt(processedText, extendedKey);

            System.out.println("The encrypted message is: " +
            encryptedText);
            System.out.println("\nDecrypting...");
            String recoveredMessage = Decrypt(encryptedText,
            extendedKey);
            System.out.println("The decrypted message is: " +
            recoveredMessage);
        }
    }
}
```

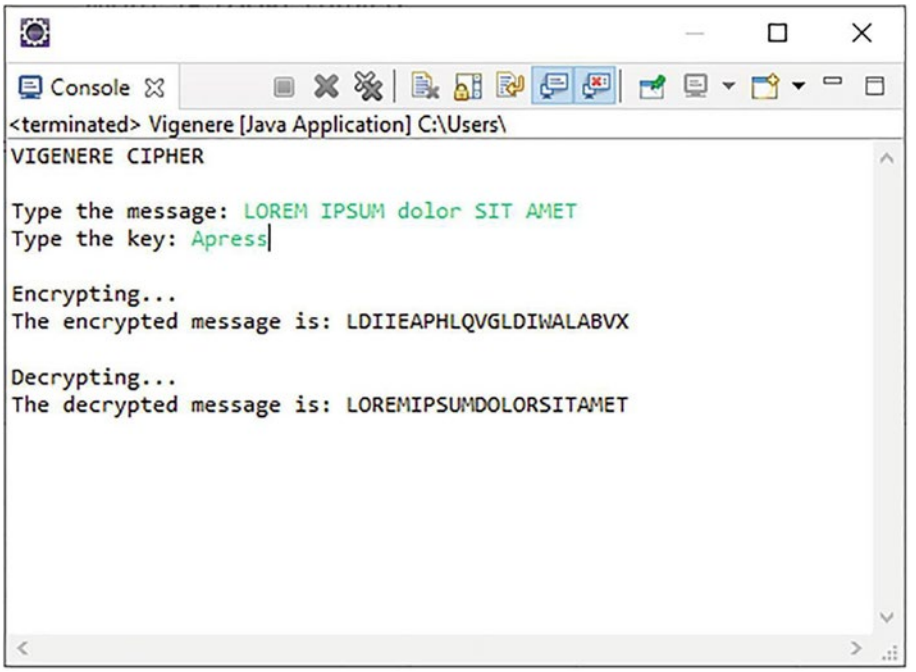


Figure 5-3. *The result of the Vigenère encryption*

Cryptanalysis

The Vigenère cipher can be broken using frequency analysis. However, it is not enough to do a simple analysis frequency because the same letter can be obtained from intersections of the plain message letter and the extended key letter. It would be easier if the length of the original key was discovered; let's denote it with n, because the encrypted text will be partitioned in blocks of length n (except for the last block that may have a smaller length). From here, the problem is reduced to breaking n Caesar ciphers. To find the key length in general, three techniques can be applied: brute-force, Kasiski examination [7], or Friedman test [8].

Hill Cipher

The author of the Hill cipher is Lester S. Hill, and it was proposed in 1929; it is an example of a polygraphic substitution cipher. The mathematical background for the Hill cipher lies in linear algebra, more exactly in matrices and their operations, in which the plaintext is hidden.

The key of the cipher is a square matrix with an arbitrary dimension, with elements in Z_{26}. For example, consider the following matrix to be the secret key,

$$K = \begin{bmatrix} 2 & 8 & 5 \\ 9 & 4 & 1 \\ 6 & 17 & 7 \end{bmatrix},$$

having three rows and three columns (denote the dimension with $n = 3$), and the plain message to be "WELCOME TO CRYPTOGRAPHY." To encrypt the message, it should proceed as follows:

1. The plain message is partitioned into blocks of three letters (because $n = 3$).
2. Convert the letters in corresponding indices of the alphabet (for convenience, see the first two rows of Table 5-1).
3. Multiply the key (square matrix) with the column matrix that represents the indices of the letters and apply modulo 26.
4. Convert the numbers into letters (for convenience, see the first two rows of Table 5-1).

Following the example, for the first three letters of the message, it will be obtained $'W' = 22$, $'E' = 4$, and $'L' = 11$, and then

$$\begin{bmatrix} 2 & 8 & 5 \\ 9 & 4 & 1 \\ 6 & 17 & 7 \end{bmatrix} \begin{bmatrix} 22 \\ 4 \\ 11 \end{bmatrix} (mod\ 26) = \begin{bmatrix} 131 \\ 225 \\ 277 \end{bmatrix} (mod\ 26) = \begin{bmatrix} 1 \\ 17 \\ 17 \end{bmatrix} \rightarrow BRR$$

Therefore, the corresponding encrypted message of "WEL" is "BRR." The process will continue until the entire message is encrypted, and for the previous example we will obtain "BRRUIWWWDAGBSRRSSNYFN." For the cases in which the last block has fewer letters than n, some extra letters will be added to the end of the plain message to achieve n letters. Note that not any matrix can be used as a key. To be invertible modulo 26, the determinant of the matrix should be coprime with 26.

To decrypt the message, the first step is to find the inverse of K (let's denote it with K') with elements also in Z_{26}; the next steps is to multiply K' with the column matrix representing the encrypted blocks. The column matrix that will be obtained represents the indices of the letters of the plain message.

$$K'(mod\ 26) = \begin{bmatrix} 21 & 1 & 22 \\ 7 & 12 & 23 \\ 17 & 22 & 22 \end{bmatrix}$$

$$\begin{bmatrix} 21 & 1 & 22 \\ 7 & 12 & 23 \\ 17 & 22 & 22 \end{bmatrix} \begin{bmatrix} 1 \\ 17 \\ 17 \end{bmatrix} (mod\ 26) = \begin{bmatrix} 412 \\ 602 \\ 765 \end{bmatrix} (mod\ 26) = \begin{bmatrix} 22 \\ 4 \\ 11 \end{bmatrix} \rightarrow WEL$$

Implementation

Listing 5-3 shows the implementation of the Hill cipher, and Figure 5-4 shows the result.

Listing 5-3. Hill Cipher Implementation

```
1   package ciphers;
2
3   import java.util.Scanner;
4
5   class Utils {
6       String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
7
8       int getCharIndex(char c) {
9           return alphabet.indexOf(c);
10      }
11
12      char getCharAtIndex(int index) {
13          return alphabet.charAt(index);
14      }
15  }
16
17  public class HillCipher {
18
19      Utils util = new Utils();
20      int keySize;
```

```
21      int[][] keyMatrix;
22      Scanner scanner = new Scanner(System.in);
23
24      HillCipher(int keySize) {
25          this.keySize = keySize;
26          this.keyMatrix = new int[keySize][keySize];
27      }
28
29      void keyInitialize(String message) {
30          System.out.println(message);
31          for (int i = 0; i < keySize; i++) {
32              for (int j = 0; j < keySize; j++) {
33                  keyMatrix[i][j] = scanner.nextInt();
34              }
35          }
36      }
37
38      String blockEncryption(String plainBlock) {
39          StringBuilder encryptedBlock = new StringBuilder();
40          int[] indexMatrix = new int[keySize];
41          int[] columnMatrix = new int[keySize];
42
43          for (int i = 0; i < keySize; i++) {
44              indexMatrix[i] = util.getCharIndex(plainBlock.charAt(i));
45          }
46
47          for (int i = 0; i < keySize; i++) {
48              int sumElem = 0;
49              for (int k = 0; k < keySize; k++) {
50                  sumElem += keyMatrix[i][k] * indexMatrix[k];
51              }
52              columnMatrix[i] = sumElem % 26;
53              encryptedBlock.append(util.getCharAtIndex(column
                Matrix[i]));
54          }
```

```
55          return encryptedBlock.toString();
56      }
57
58      String messageEncryption(String plainMessage) throws Exception {
59          StringBuilder encryptedMessage = new StringBuilder();
60          keyInitialize("Enter the values for key matrix:");
61          System.out.println("\n[Encrypting...]\n");
62
63          while (plainMessage.length() % keySize != 0) {
64              plainMessage += "Z";
65          }
66
67          for (int i = 0; i < plainMessage.length(); i += keySize) {
68              encryptedMessage.append(blockEncryption(plainMessage.
                substring(i, i + keySize)));
69          }
70          return encryptedMessage.toString();
71      }
72
73      String blockDecryption(String encryptedBlock) {
74          StringBuilder decryptedBlock = new StringBuilder();
75          int[] indexMatrix = new int[keySize];
76          int[] columnMatrix = new int[keySize];
77
78          for (int i = 0; i < keySize; i++) {
79              indexMatrix[i] = util.getCharIndex(encryptedBlock.
                charAt(i));
80          }
81
82          for (int i = 0; i < keySize; i++) {
83              int sumElem = 0;
84              for (int k = 0; k < keySize; k++) {
85                  sumElem += keyMatrix[i][k] * indexMatrix[k];
86              }
```

```
87              sumElem = (sumElem % 26 + 26) % 26;  // Normalize
                                                     negative values
88              columnMatrix[i] = sumElem;
89              decryptedBlock.append(util.getCharAtIndex(column
                Matrix[i]));
90          }
91          return decryptedBlock.toString();
92      }
93
94      String messageDecryption(String encryptedMessage) throws
        Exception {
95          StringBuilder decryptedMessage = new StringBuilder();
96          keyInitialize("\n---\n\n Enter the values for inverted key
            matrix:");
97          System.out.println("\n[Decrypting...]\n");
98
99          for (int i = 0; i < encryptedMessage.length(); i += keySize) {
100             decryptedMessage.append(blockDecryption(encryptedMessage.
                substring(i, i + keySize)));
101         }
102         return decryptedMessage.toString();
103     }
104
105     public static void main(String[] args) throws Exception {
106         System.out.println("Enter the plain message:");
107         String plainMessage = new Scanner(System.in).nextLine();
108
109         System.out.println("Enter the size of the matrix key:");
110         int keySize = new Scanner(System.in).nextInt();
111
112         HillCipher cipher = new HillCipher(keySize);
113
114         plainMessage = plainMessage.replaceAll(" ", "").toUpperCase();
115
```

```
116         String encryptedMessage = cipher.messageEncryption
            (plainMessage);
117         System.out.println("Obtained encrypted message: \n" +
            encryptedMessage);
118
119         String decryptedMessage = cipher.messageDecryption(encrypted
            Message);
120         System.out.println("Obtained decrypted message:\n" +
            decryptedMessage);
121     }
122 }
```

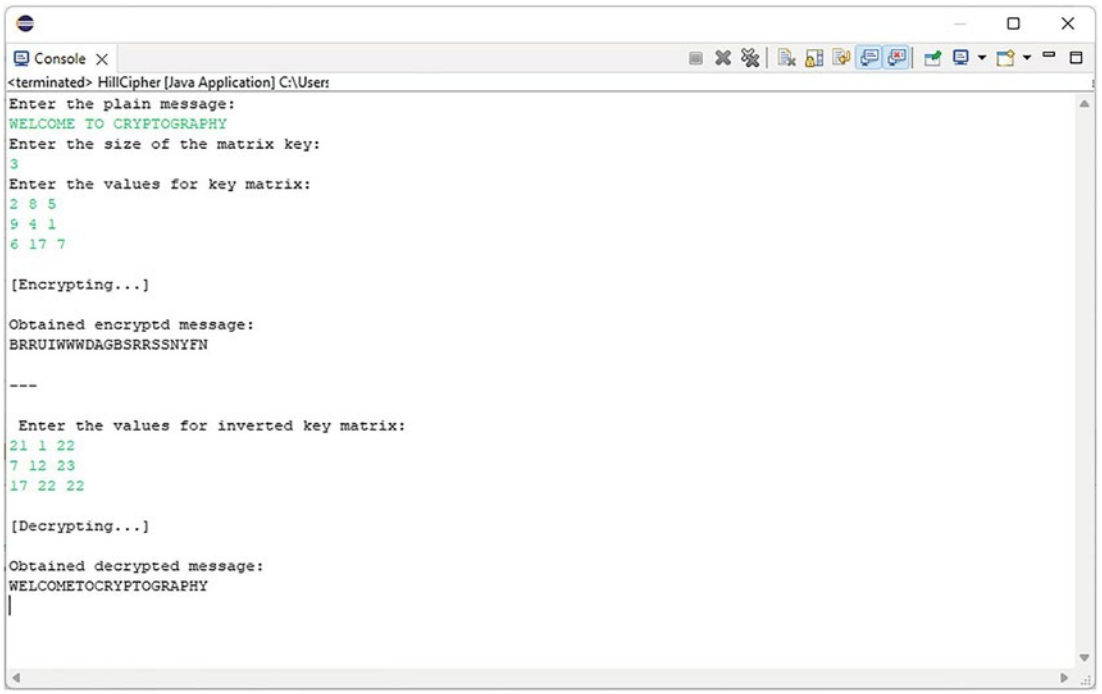


Figure 5-4. *The result of Listing 5-3*

The previous code follows the steps of the Hill cipher implementation, but it is a little more difficult than previous ones. First, the plain message/encrypted message is split into blocks of the key's length. Then, each block is encrypted and added to a decrypted message/plain message. The method `blockEncryption(String plainBlock)` converts each character in the `plainBlock` to a numeric index, performs matrix multiplication

with the `keyMatrix`, and converts the resulting matrix back into text. Each character of the encrypted block is determined by the modular arithmetic on these indices. The method `messageEncryption(String plainMessage)` ensures the message length is a multiple of the `keySize` by padding if necessary and then calls `blockEncryption` for each block and concatenates the results. The method `blockDecryption(String encryptedBlock)` is similar to `blockEncryption` but uses the inverted key matrix (or an equivalent process to achieve decryption). It reverses the encryption process to recover the original text block. Lastly, the `messageDecryption(String encryptedMessage)` method is similar with `messageEncryption` but calls `blockDecryption` for each block. It processes the entire encrypted message, decrypting it block by block and assembling the decrypted message. The encryption and decryption processes are exemplified in the `main` method.

Other good references for Java implementations of the Hill cipher are [10] and [11].

Cryptanalysis

A known-plaintext attack can easily break the Hill cipher; therefore, if the attacker knows the plaintext and the corresponding encrypted message, the key can be recovered immediately, since all computations are linear. For a 2×2 key, the attack consists of computing the frequency for each digraph that may be obtained in the encrypted text. To launch this attack, knowing the standard English digraph representations is mandatory. For example, the group of letters *th* and *he* are common. A good resource for Hill cipher cryptanalysis is [9].

Conclusion

This chapter showed that primordial forms of cryptography existed in ancient times and presented three of the most important historical ciphers. These are considered symmetric ciphers because the same key is used for both encryption and decryption and it should be secret, known only by the sender and by the receiver. Caesar, Vigenère, and Hill are simple ciphers from the technical perspective and are a good starting point for understanding cryptography. However, these ciphers are not used in practice because are easily cracked.

References

[1] Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A. (2018). Handbook of applied cryptography. CRC Press.

[2] Paar, C., & Pelzl, J. (2009). Understanding cryptography: a textbook for students and practitioners. Springer Science & Business Media.

[3] Stanoyevitch, A. (2010). Introduction to Cryptography with mathematical foundations and computer implementations. CRC Press.

[4] Practical Cryptography (website). Quadgram Statistics as a Fitness Measure, http://practicalcryptography.com/cryptanalysis/text-characterisation/quadgrams/.

[5] Practical Cryptography (website). Cryptanalysis of Caesar Cipher, http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-caesar-cipher/.

[6] Mihailescu, M. I., & Nita, S. L. (2021). Cryptography and Cryptanalysis in MATLAB. Apress, Berkeley, CA.

[7] Kasiski, F. W. (1863). Die Geheimschriften und die Dechiffrir-Kunst: mit besonderer Berücksichtigung der deutschen und der französischen Sprache. ES Mittler und sohn.

[8] Van Tilborg, H. C., & Jajodia, S. (Eds.). (2014). Encyclopedia of cryptography and security. Springer Science & Business Media.

[9] Practical Cryptography (website). Hill Cipher, http://practicalcryptography.com/ciphers/classical-era/hill/.

[10] Hill Cipher Program in Java, https://www.javatpoint.com/hill-cipher-program-in-java.

[11] Hill-Cipher, https://github.com/tstevens/Hill-Cipher/blob/master/Main.java.

CHAPTER 6

Quantum Cryptography

Quantum cryptography is an approach to securing communication through the principles of quantum mechanics, fundamentally different from traditional cryptographic methods, which are based on mathematical complexity. Here's a beginner-friendly introduction to this fascinating field.

Quantum cryptography uses the principles of quantum physics to implement cryptographic functions, with an important use case being the secure exchange of encryption keys. It is based on the rules of physics for security, rather than the difficulty of mathematical problems like classical cryptography, making it highly secure [1]–[3].

Key Concepts for Quantum Mechanics Used in Cryptography

This section provides an overview of the most important components in quantum cryptography based on information from [1]–[8].

Quantum Superposition

Quantum systems have the ability to exist in several states simultaneously until a measurement is taken. In cryptography, a quantum bit (qubit) can simultaneously represent both 0 and 1. Creating a true quantum superposition in Java is not feasible due to Java's foundation in classical computing. We can simulate superposition for educational purposes to understand how states might behave in a quantum system. We will achieve this by constructing a simple model in which a "qubit" can exist in a superposition of the states 0 and 1.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_6

The concept of quantum superposition. A quantum bit or qubit in quantum mechanics can exist in a superposition state of both 0 and 1 simultaneously, unlike a classical bit that is strictly either 0 or 1. This superposition is typically represented as $\psi = \alpha \mid 0\rangle + \beta \mid 1\rangle$ where α and β are complex numbers (C) that describe the probability amplitudes of the qubit being in the 0 or 1 state, respectively.

Simulating quantum superposition in Java programming language. We'll simulate a qubit in superposition by representing αα and β as floating-point numbers. To simplify, we will restrict these to real numbers instead of complex numbers. We will design a class called `Qubit` that stores these probabilities and can transition to a specific state (0 or 1) when observed, based on their individual probabilities. Listing 6-1 shows the implementation for this simulation, and Figure 6-1 shows the result.

Listing 6-1. Simulation of a Qubit

```
import java.util.Random;

public class Qubit {
    private double alpha; // Amplitude for state |0>
    private double beta;  // Amplitude for state |1>
    private static final Random random = new Random();

    // Constructor to initialize the qubit in superposition
    public Qubit(double alpha, double beta) {
        double norm = Math.sqrt(alpha * alpha + beta * beta);
        this.alpha = alpha / norm;
        this.beta = beta / norm;
    }

    // Method to "measure" the qubit, collapsing its superposition
    public int measure() {
        // Random number between 0 and 1
        double p = random.nextDouble();
        return p < alpha * alpha ? 0 : 1;
    }

    public static void main(String[] args) {
```

```
23          // Example: Superposition close to even, alpha=1, beta=1
24          Qubit qubit = new Qubit(1, 1);
25          // Measure the qubit multiple times to see the
            probabilistic outcome
26          int zeroCount = 0, oneCount = 0;
27          int trials = 1000;
28          for (int i = 0; i < trials; i++) {
29              if (qubit.measure() == 0) {
30                  zeroCount++;
31              } else {
32                  oneCount++;
33              }
34          }
35          System.out.println("Measured '0': " + zeroCount + " times");
36          System.out.println("Measured '1': " + oneCount + " times");
37      }
38  }
```

The code from Listing 6-1 contains the following steps:

- **The initialization of qubit (lines 8–11).** The qubit is set to a superposition state determined by the amplitudes for states |0> and |1>. These amplitudes are normalized to guarantee they represent legitimate probability amplitudes.
- **Measurement (lines 16–20).** When measuring the qubit, a random number is utilized to determine if the qubit will collapse. When measuring the qubit, a random number is utilized to determine if the qubit will collapse to state |0> or |1> based on the probability derived from the square of the amplitude. This is compliant with the rules of quantum mechanics.
- **Simulation (lines 25–31).** We repeatedly measure the qubit to observe the distribution of events, which should correspond to the probabilities indicated by the superposition state.

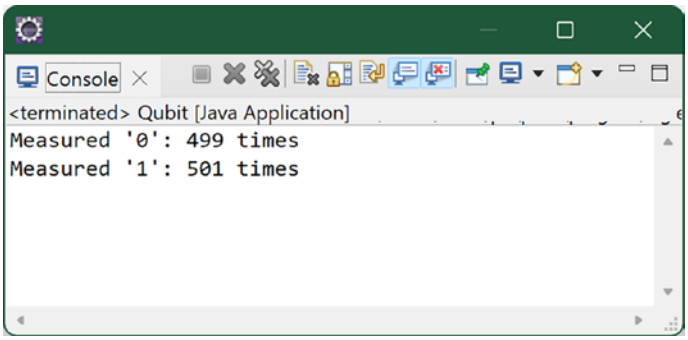


Figure 6-1. *The result for Listing 6-1*

Quantum Entanglement

Particles like photons can become entangled, causing the state of one to directly influence the state of the other, regardless of the distance between them. This characteristic is capable of identifying eavesdropping.

Quantum entanglement occurs when particles become coupled in a manner where the state of one particle cannot be described independently of the state of the other, regardless of the distance between them. This fundamental principle in quantum mechanics is not immediately applicable in conventional computers. We may simulate some aspects of entanglement with classical programming methods to analyze its conceptual features, similar to superposition. Further, Listing 6-2 shows a basic simulation of quantum entanglement in Java, focusing on a simple scenario involving two qubits. This example will demonstrate the behavior of a *bell state*, which is a quantum state that includes two entangled qubits. To simplify, we will use a basic model where the result of measuring one qubit directly influences the result of measuring the other. Figure 6-2 shows the result of running the algorithm.

Listing 6-2. Simulating a Quantum State

```
1   import java.util.Random;
2
3   public class EntangledQubits {
4       private boolean stateA;  // State of qubit A
5       private boolean stateB;  // State of qubit B
6
```

```
7       // Flag to check if measurement has occurred
8       private boolean measured = false;
9
10      private static final Random random = new Random();
11
12      // Constructor for entangled qubits
13      public EntangledQubits() {
14          // Initialize the entangled pair
15          stateA = random.nextBoolean();
16
17          // Ensures they are opposite for a specific Bell state |Ψ+> =
            (|01⟩ + |10⟩)/√2
18          stateB = !stateA;
19      }
20
21      // Method to measure qubit A
22      public int measureA() {
23          if (!measured) {
24              measured = true;
25          }
26          return stateA ? 1 : 0;
27      }
28
29      // Method to measure qubit B
30      public int measureB() {
31          if (!measured) {
32              measured = true;
33
34              // Ensure state B is the opposite of A when measured first
35              stateB = !stateA;
36          }
37          return stateB ? 1 : 0;
38      }
39
40      public static void main(String[] args) {
```

```
41          EntangledQubits entangledQubits = new EntangledQubits();
42
43          int measurementA = entangledQubits.measureA();
44          int measurementB = entangledQubits.measureB();
45
46          System.out.println("Measurement of Qubit A: " + measurementA);
47          System.out.println("Measurement of Qubit B: " + measurementB);
48      }
49  }
```

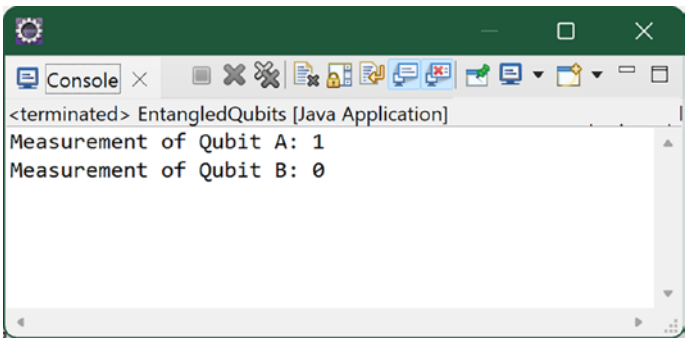


Figure 6-2. *The result of running Listing 6-2*

The code from Listing 6-2 works as follows:

- **Entangled pair initialization.** When created, one qubit (A) randomly selects between true and false. The state of qubit B is directly set to be the opposite. This configuration replicates a precise bell state referred to as $|\Psi+\rangle$, forcing the qubits to be in opposing states.
- **Measurement methods.** Every qubit possesses a technique to determine its state. When a qubit is measured first, the `measured` flag is set to true, influencing the subsequent measurement of the other qubit to maintain its correlation as required by entanglement.
- **Simulation.** The program emulates measuring both qubits and displays the results, demonstrating the expected correlation in an entangled state.

Note that the simulation is a simplified classical representation that does not capture the complexity of genuine quantum phenomena, such as probabilities and

superpositions that classical binary states cannot accurately replicate. It offers a helpful conceptual connection to explain how entanglement could impact the measurement results of coupled quantum systems.

Heisenberg's Uncertainty Principle

Any attempt to observe or measure a quantum system will inevitably cause a change in its state. This idea is essential in quantum cryptography as it guarantees that any effort to intercept quantum transmissions will be detected.

The Heisenberg Uncertainty Principle is an essential notion in quantum physics that states the impossibility of knowing both the exact position and momentum of a particle simultaneously. It suggests that specific pairs of attributes, called *complementary variables*, cannot be accurately measured or known simultaneously.

Because of Java's deterministic nature as a classical programming language, it is not feasible to accurately replicate the behavior of quantum events such as the Heisenberg Uncertainty Principle. However, we may replicate a scenario that theoretically embodies the essence of this idea by creating a basic class that imitates uncertainty.

Listing 6-3 builds a scenario where measuring either the position or momentum of a "quantum particle" (in this case, a simulated object) impacts the certainty of the other characteristic. This phenomenon can be represented by random oscillations that increase as a specific property is measured with greater precision. Figure 6-3 shows The result.

Listing 6-3. Measuring Properties of a "Quantum Particle"

```
import java.util.Random;

public class QuantumParticle {
    private double position;
    private double momentum;
    private static final Random random = new Random();

    public QuantumParticle(double position, double momentum) {
        this.position = position;
        this.momentum = momentum;
    }
```

```
12
13      // Simulates measuring the position
14      public double measurePosition() {
15          double uncertainty = 0.1; // Base uncertainty
16          double fluctuation = random.nextDouble() * uncertainty;
17          // Introduce fluctuation in momentum when position is measured
18          momentum += (random.nextBoolean() ? 1 : -1) * fluctuation;
19          return position + (random.nextBoolean() ? 1 : -1) * fluctuation;
20      }
21
22      // Simulates measuring the momentum
23      public double measureMomentum() {
24          double uncertainty = 0.1; // Base uncertainty
25          double fluctuation = random.nextDouble() * uncertainty;
26          // Introduce fluctuation in position when momentum is measured
27          position += (random.nextBoolean() ? 1 : -1) * fluctuation;
28          return momentum + (random.nextBoolean() ? 1 : -1) * fluctuation;
29      }
30
31      public static void main(String[] args) {
32          QuantumParticle particle = new QuantumParticle(5.0, 10.0);
33          System.out.println("Initial position: " + particle.position);
34          System.out.println("Initial momentum: " + particle.momentum);
35
36          double measuredPosition = particle.measurePosition();
37          double measuredMomentum = particle.measureMomentum();
38
39          System.out.println("Measured position: " + measuredPosition);
40          System.out.println("Measured momentum: " + measuredMomentum);
41          System.out.println("Updated position: " + particle.position);
42          System.out.println("Updated momentum: " + particle.momentum);
43      }
44  }
```

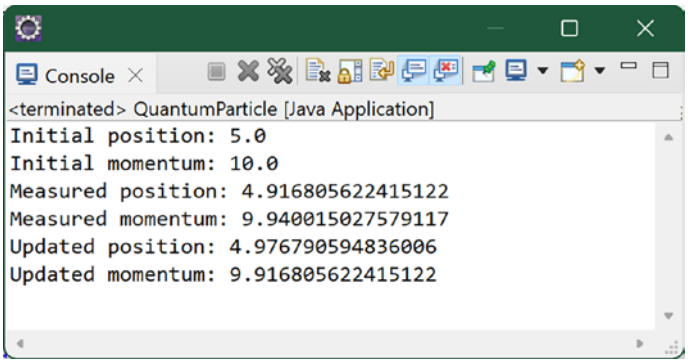


Figure 6-3. *The result for Listing 6-2*

The code works as follows:

- **Initialization (lines 8–9).** A `QuantumParticle` is initialized with specific values for position and momentum.
- **Measurement functions (lines 14–29).** The `measurePosition()` and `measureMomentum()` methods introduce random fluctuations to the complementary variable (`momentum` and `position`, respectively). This simulates the uncertainty principle by showing that trying to measure one property more precisely affects the certainty of the other property.
- **Simulation output (lines 36 and 37).** When the position is measured first, and then the momentum, both values are reported along with the resulting updated state of the particle. It can be seen how the initial precise values are no longer maintained.

Note this program is a classical simulation and cannot accurately represent quantum behavior, where wave functions and operators symbolize properties such as position and momentum in a complex Hilbert space. It acts as a conceptual tool to demonstrate the uncertainty and interconnection among quantum phenomena in an educational or illustrative context.

Quantum Key Distribution

Quantum key distribution (QKD) is the most notable use of quantum cryptography, allowing two parties to share a cryptographic key securely. The BB84 protocol, proposed by Charles Bennett and Gilles Brassard in 1984, is the first and notable QKD system [9]. It has the following structure:

- **Key creation.** Alice generates a random sequence of bits and assigns each bit to a photon by randomly selecting one of two polarization bases for encoding. The shapes can be either rectilinear (vertical or horizontal) or diagonal (at 45 degrees or 135 degrees).
- **Transmission.** Alice sends these photons to Bob over a quantum channel (like an optical fiber).
- **Measurement.** Bob measures the incoming photons using randomly chosen bases. To successfully decode the bit, he must select the same base that Alice used to encode it, as required by the principles of quantum mechanics..
- **Basis reconciliation.** Alice and Bob publicly compare their bases. They remove any bits where their bases did not match, as these bits are unreliable.
- **Key generation.** The remaining bits form the secret key.

Security and Eavesdropping Detection

One important characteristic of QKD is its inherent capability to identify eavesdropping [10]. Eve, attempting to intercept the key by measuring the quantum states, will disturb the quantum state of the photons due to the Heisenberg Uncertainty Principle. This will introduce noticeable flaws in the key bits transferred between Alice and Bob. Alice and Bob can assess the security of their key and detect interception attempts by analyzing a portion of the key and calculating the error rate.

Challenges

Quantum cryptography has potential but encounters challenges. Sophisticated and costly quantum technologies are now necessary, and the range of key distribution is restricted by photon loss in optical fibers or open space. Advancements in quantum repeaters and satellite-based quantum communication are leading toward a future where quantum cryptography may be extensively utilized.

Quantum cryptography is a strong security measure that may be essential in the age of quantum computing, as conventional encryption methods could be at risk. This is a dynamic field of research that has the potential to change the data security and privacy environment greatly.

BB84 Protocol

The BB84 protocol [9] introduced the concept of QKD, which leverages the principles of quantum mechanics to allow two parties to securely generate a shared secret key, even in the presence of an eavesdropper. The key generated can then be used to encrypt and decrypt messages using classical cryptographic techniques. These are the components and steps involved in the BB84 protocol:

1. **Quantum and classical channels.** BB84 requires two distinct communication channels between the two parties, generally known as Alice and Bob: a *secure quantum channel* is used to transmit individual quantum bits (qubits), while an *insecure classical channel* is used for public communication that does not need to be kept secret.
2. **Encoding information in qubits.** The protocol uses polarization or phase encoding of qubits to symbolize binary values (0s and 1s). The qubits are encoded using two sets of orthogonal bases. Usually, these are as follows:
    - The rectilinear basis, where 0 and 1 are represented by two perpendicular polarization directions (e.g., vertical and horizontal).
    - The diagonal basis, where 0 and 1 are represented by two diagonal polarization directions (e.g., 45 degrees and 135 degrees).

3. **Preparing and sending qubits.** Alice chooses a bit value (0 or 1) and a basis (rectilinear or diagonal) for each qubit and then transmits it to Bob over the quantum channel.
4. **Measuring qubits.** Upon receiving each qubit, Bob randomly selects a basis to measure it. Bob will accurately determine the bit value only if he selects the identical basis that Alice employed to encode the bit, as a result of quantum physics features.
5. **Basis reconciliation.** Alice and Bob utilize the classical channel to share and compare the bases they used for each qubit after sending and measuring all qubits, without revealing the bit values. They eliminate any segments where their foundations did not align.
6. **Error detection and privacy amplification.** Alice and Bob perform error detection and correction, as well as privacy amplification, to protect the key's integrity and prevent eavesdropping. It includes publicly comparing a portion of their corresponding key bits. If the error rate is acceptable, they then utilize privacy amplification methods to reduce the key's length, thus increasing its security.
7. **Key generation.** The remaining, unrevealed bits where the bases matched and no errors were found form the secret key.

Security Based on Quantum Principles

The security of the BB84 protocol is fundamentally based on the principles of quantum mechanics [10].

- **No-cloning theorem.** Quantum information is not reproduced without altering the initial state. An eavesdropper will make detectable flaws in the bit values if she tries to intercept and measure the qubits.
- **Heisenberg uncertainty principle.** Measuring a quantum state on the wrong basis results in random outcomes, which helps maintain the confidentiality of accurately aligned measurements.

The quantum effects guarantee that any effort to eavesdrop on the quantum channel may be identified, enabling Alice and Bob to confirm the security of their key distribution protocol. The BB84 protocol was a breakthrough advancement in quantum communication and continues to be a fundamental element in the research and implementation of quantum cryptography. Listing 6-4 shows a basic implementation of BB84 protocol, while Figure 6-4 shows the result.

Description of BB84 Protocol Simulation

Here is what happens in the simulation:

1. **Photon generation.** Alice creates a random sequence of bits and encodes them into photons, randomly selecting between two bases, such as rectilinear or diagonal.
2. **Photon transmission.** Alice sends these photons to Bob.
3. **Photon measurement.** Bob randomly chooses bases to measure the received photons.
4. **Basis reconciliation.** Alice and Bob communicate publicly to discard the bits where they used different bases.
5. **Key generation.** The remaining bits form the secret key.

Listing 6-4. Simulation of BB84 Protocol

```
import java.util.Random;

public class BB84 {
    private static final Random random = new Random();

    // Method to generate a random binary string of a given length
    public static String generateRandomBits(int length) {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < length; i++) {
            result.append(random.nextBoolean() ? '1' : '0');
        }
        return result.toString();
```

```
    }

    // Method to simulate the key exchange
    public static String keyExchange(String bits, String aliceBases,
    String bobBases) {
        StringBuilder key = new StringBuilder();
        for (int i = 0; i < bits.length(); i++) {
            // Keep the bit if Alice and Bob used the same base
            if (aliceBases.charAt(i) == bobBases.charAt(i)) {
                key.append(bits.charAt(i));
            }
        }
        return key.toString();
    }

    public static void main(String[] args) {
        int length = 100; // Length of the bit string
        String aliceBits = generateRandomBits(length);
        String aliceBases = generateRandomBits(length);
        String bobBases = generateRandomBits(length);

        // Alice sends bits, Bob measures them
        String finalKey = keyExchange(aliceBits, aliceBases, bobBases);

        System.out.println("Alice's original bits: " + aliceBits);
        System.out.println("Alice's bases:        " + aliceBases);
        System.out.println("Bob's bases:          " + bobBases);
        System.out.println("Generated Key:        " + finalKey);
    }
}
```

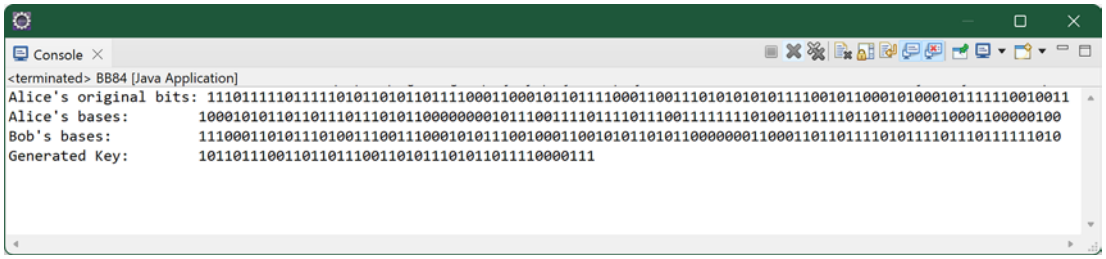


Figure 6-4. *The result of simulating the BB84 protocol*

The main steps in Listing 6-4 are random bit generation (lines 7–14), where both Alice and Bob generate random strings for their bits and the bases they choose to encode or measure those bits and key exchange (lines 16–25), where they compare their bases and keep the bits where their bases match. This example is greatly simplified and does not incorporate any real quantum mechanics. A genuine quantum key distribution system would include qubits, quantum channels, and physical components such as photon detectors and sources. This simulation is purely for conceptual demonstration purposes.

Other Quantum Cryptography Algorithms

The BB84 protocol is a quantum cryptography technique that primarily focuses on QKD. Different protocols and algorithms have been proposed to enhance or offer substitutes for BB84. These include multiple versions of QKD and alternative methods of leveraging quantum physics for cryptographic objectives. The following are other quantum cryptography algorithms [2]–[7]:

- **E91 (Ekert protocol).** Artur Ekert created this protocol in 1991, which utilizes quantum entanglement to distribute securely a key between two parties. The security is ensured against eavesdropping by utilizing the Einstein-Podolsky-Rosen (EPR) pair phenomenon and Bell's inequality tests. This protocol both creates a key and validates the security of the transmitted data at the same time.
- **B92 (Bennett 1992 protocol).** Another protocol developed by Charles Bennett in 1992, the B92 uses a simpler scheme than BB84, requiring only two nonorthogonal quantum states. Simplifying the protocol may make implementations easier, but it could also reduce the system's resilience to specific types of attacks when compared to BB84.

- **Six-State protocol.** The Six-State protocol is an extension of BB84 that use three bases and six distinct states of the qubits. The expansion of states enhances security by increasing the amount of data available to identify eavesdropping attempts on the quantum channel, specifically regarding noise and probable manipulation.
- **Differential phase shift (DPS) QKD.** This technique utilizes a series of phase-encoded pulses, as opposed to BB84, which encodes information in individual photons. The key is encoded in the relative phases of consecutive pulses rather than the state of a single photon, providing various security features and efficiency.
- **Continuous variable QKD.** Continuous variable QKD utilizes continuous quantum variables like the quadratures of the electromagnetic field, in contrast to QKD algorithms that typically rely on discrete variables. Implementing this protocol using normal telecommunication technologies may decrease the expenses and complexity of quantum cryptography configurations.
- **Measurement-device independent QKD (MDI-QKD).** MDI-QKD was created to address security vulnerabilities associated with detector side-channel attacks. It requires both parties to communicate quantum states to an untrusted relay, which then does a joint measurement. This relay is unable to acquire the key but can assist the parties in safely establishing one, even in the presence of an attacker who has complete control over the measuring devices.
- **Twin field QKD (TF-QKD).** A novel development in QKD enables significantly greater transmission distances than conventional techniques. Weak coherent pulses are transmitted to a central node for a Bell-state measurement. This protocol is recognized for its ability to greatly extend the effective range of secure quantum communications.
- **Quantum secure direct communication (QSDC).** QSDC protocols provide the direct transmission of secret messages without requiring a pre-existing shared secret key, in addition to key distribution. This is accomplished by using the technique of quantum entanglement and the transfer of quantum states, in which the message is encoded directly into the quantum states.

Each protocol uses several elements of quantum mechanics, such as the no-cloning theorem, entanglement, and the uncertainty principle, to ensure security. Quantum cryptography is a rapidly evolving topic, with research focused on enhancing the practicality and accessibility of these technologies for real-world use.

Conclusions

The chapter introduced quantum cryptography, focusing on fundamental concepts such as quantum superposition, entanglement, and the Heisenberg Uncertainty Principle, which are essential in quantum encryption. We used Java simulations to simulate quantum phenomena in a classical programming environment, but actual quantum effects can be achieved only with quantum hardware. The chapter discussed quantum key distribution, focusing on the BB84 protocol, highlighting its innovative method for secure communication that is naturally resistant to quantum attacks. With the progression of quantum computing, it is becoming more important to comprehend and get ready for the integration of quantum-resistant cryptography.

References

[1] M. Hirvensalo. Quantum computing. Springer Science & Business Media, 2013.

[2] V. Padamvathi, B. V. Vardhan and A. V. N. Krishna, "Quantum Cryptography and Quantum Key Distribution Protocols: A Survey," 2016 IEEE 6th International Conference on Advanced Computing (IACC), Bhimavaram, India, 2016, pp. 556-562, doi: 10.1109/IACC.2016.109.

[3] M. Ati, "Implementation of Quantum Cryptography for Securing IoT Devices," 2023 International Conference on Electrical, Communication and Computer Engineering (ICECCE), Dubai, United Arab Emirates, 2023, pp. 1-5, doi: 10.1109/ICECCE61019.2023.10442918.

[4] J. Moazzam, R. Pawar and M. D. Khare, "Evolution and Advancement of Quantum Computing in the Era of Networking and Cryptography," 2023 International Conference on Advances in Computation, Communication and Information Technology (ICAICCIT), Faridabad, India, 2023, pp. 817-821, doi: 10.1109/ICAICCIT60255.2023.10465946.

[5] C. R. Garcia, A. C. Aguilera, J. J. V. Olmos, I. T. Monroy and S. Rommel, "Quantum-Resistant TLS 1.3: A Hybrid Solution Combining Classical, Quantum and Post-Quantum Cryptography," 2023 IEEE 28th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Edinburgh, United Kingdom, 2023, pp. 246-251, doi: 10.1109/CAMAD59638.2023.10478407.

[6] P. Bhale, D. Padulkar and J. Abraham, "Binary Voting Protocol Using Quantum Secret Sharing," 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), Bangalore, India, 2023, pp. 1-5, doi: 10.1109/CSITSS60515.2023.10334245.

[7] A. Takalkar and B. Shiragapur, "Quantum Cryptography: Mathematical Modelling and Security Analysis," 2023 3rd Asian Conference on Innovation in Technology (ASIANCON), Ravet IN, India, 2023, pp. 01-07, doi: 10.1109/ASIANCON58793.2023.10270593.

[8] P. Naing, K. Z. Oo and M. M. Su Thwin, "Proposed Security Enhancement Conceptual Models Using Quantum Key Distribution for Future Cryptography," 2023 IEEE Conference on Computer Applications (ICCA), Yangon, Myanmar, 2023, pp. 399-404, doi: 10.1109/ICCA51723.2023.10181459.

[9] C. H. Bennett and G. Brassard, in Proceedings of IEEE International Conference on Computers, Systems, and Signal Processing, Bangalore, India (IEEE, New York, 1984), p. 175.

[10] Shor, P. W., & Preskill, J. (2000). Simple proof of security of the BB84 quantum key distribution protocol. Physical review letters, 85(2), 441.

CHAPTER 7

Formal Techniques for Cryptography

This chapter discusses mathematical elements used in cryptography and throughout the remainder of this book. We'll start with some definitions of specific terminology; then, we'll cover some minimal notions of algebra; and finally, we'll delve into elliptic curves. This chapter contains a compilation of different technical (mathematical) tools used in cryptography gathered from different resources [1]–[9].

Definitions

Here are some definitions you need to know:

- **Sets.** The usual sets of numbers are $\mathbb{N}$ (natural numbers set), $\mathbb{Z}$ (integer numbers set), $\mathbb{R}$ (real numbers set), and $\mathbb{C}$ (complex numbers set). Other important sets used in cryptography are $\mathbb{Z}_n = \frac{\mathbb{Z}}{n\mathbb{Z}} = \{0,\ldots,n-1\}, n \geq 2$ (set of integers modulo integer n). Given a prime number p, the field $\mathbb{F}_p = \mathbb{Z}_p$. Lastly, $\mathcal{P}(S)$ contains the set of all subsets of S (where S is itself a set).
- **Bits.** An element of the set $\mathbb{Z}_2 = \{0, 1\}$ is called a bit. A *bit string* of length n is a vector that contains n bits. The notation $\{0, 1\}^n$ represents the set of bit strings of length n. To represent the set of all bit strings of finite length (including the empty string), the notation $\{0, 1\}*$ is used.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_7

- **Cryptographic primitive.** This represents "easier" algorithms used to construct cryptographic protocols or other complex cryptographic mechanisms. For example, the one-way hash function is a cryptographic primitive.
- **Cryptosystem.** This is a tuple of the form ϵ = (*KeyGeneration*, *Encryption*, *Decryption*) that consists of three probabilistic polynomial-time algorithms, i.e., the key generation algorithm, the encryption algorithm, and the decryption algorithm, that must fulfill the following condition:

$$Dec_{K'}\left(Enc_{K}\left(m\right)\right)=m,$$

  where K is the encryption key, K' is the decryption key, m is the plain message, *Enc* is the encryption algorithm, and *Dec* is the decryption algorithm. Note that according to the type of encryption, K may be the same as K', in which case it should be kept private, or K may be different from K', in which case K is public and K' is secret. You can find more about cryptosystems types in Chapter 10 and Chapter 11.
- **Hardness assumption.** This represents a computational assumption that states a problem cannot be solved efficiently (by efficiently it means in polynomial time regarding a quantity). Some common examples of hardness assumptions are integer factorization, discrete logarithm, cryptographic pairings, Diffie-Hellmann assumptions, etc.
- **Protocol.** Usually, a protocol is applied between n entities and is a procedure with a set of rules through which messages are exchanged between the n entities. To construct messages, public sets and/or private sets are used based on cryptographic primitives. For message exchange, it can be required sequential communication.
- **Authorities and third parties.** Authorities are usually used to issue digital certificates, while trusted third parties are intermediary entities for reciprocal actions between two entities in which they both trust each other.

- **Attacks.** In cryptography, an attack is a technique that puts in danger the security of a system by searching for security weaknesses of the system and exploiting them for malicious purposes. The same definition applies to *cryptanalysis*, but the major difference between an attack and cryptanalysis is its purpose. The goal of cryptanalysis is to find the security weaknesses of a system, but the goal is to overcome those security issues and to provide improvements or security alternatives to protect the system.

Probabilities and Statistics

In cryptography, statistics and probabilities have an important role, especially in the security analysis process for cryptosystems, and in a branch of cryptography called *steganalysis*. This section covers basic statistics and probabilities notions. For more advanced concepts, you can consult [9]–[11].

Let's start by defining the notion of experiment. An *experiment* is a procedure through which a value is obtained from a predefined set of values $S = \{s_1, \ldots, s_n\}$, called a *sample space*. A sample is an individual value within the given set S. A specific value from S is called an *event*. A *discrete* sample space is a set for which the number of possible outcomes is limited.

A probability distribution over the sample space S is a set $K = \{k_1, \ldots, k_m\}$, with $K \subset S$, for which $k_1 + k_2 + \ldots + k_m = 1$. The value k_i is the probability that the event s_i will occur.

The probability of an event E is denoted with $P(E)$. When the event E contains values from S, $E = \{e_1, \ldots, e_k\}$, the probability of the event E is computed as $P(E) = P(e_1) + \ldots + P(e_k)$. The complementary event of E is denoted with $\overline{E}$ and means all values that do not belong to E.

The following relations occur between S, E, and $\overline{E}$, where $E \subset S$:

$0 \leq P(E) \leq 1$.

$P(E) = 0$ if $E = \phi$ (empty set); $P(E) = 1$ if $E = S$.

If every element of S has the same probability of occurring, then $P(E) = \frac{\#E}{\#S}$ (where # is the cardinality of the set that follows the symbol).

$$P\left(\bar{E}\right)=1-P\left(E\right).$$

Two events, E_1, E_2, are called mutually exclusive when $P(E_1 \bigcap E_2) = 0$.

Conditional Probability

Conditional probability occurs in situations in which one event depends on one or more other events. Let E_1 and E_2 be two events with $P(E_2) > 0$. The conditional probability for E_1 to occur conditioned by E_2 is denoted with $P(E_1| E_2)$ and has the following formula:

$$P\left(E_1|E_2\right)=\frac{P\left(E_1\cap E_2\right)}{P\left(E_2\right)}$$

Put more simply, this can be seen as how likely E_1 will occur if E_2 has already occurred.

Regarding conditional probability, the following relationships take place:

- $P(E_1 \bigcap E_2) = P(E_1) \cdot P(E_2)$
- Bayes theorem

$$P\left(E_1|E_2\right)=\frac{P\left(E_1\right)\cdot P\left(E_2|E_1\right)}{P\left(E_2\right)}, P\left(E_2\right)>0$$

An important remark here is that $P(E_1| E_2)$ is different from $P(E_2| E_1)$. In the first relation, E_1 is conditioned by E_2, while in the second relation E_2 is conditioned by E_1.

Random Variables

Let S be a sample space with distribution probability P, and let X be a function that maps S to $\mathbb{R}$. For each element $s_i \in S$, a value $x_i = X(s_i)$ will exist. In this setting, X is called a *random variable*. A random variable gives two important notions in statistics: mean (or expected value) and variance. These values can be computed using these formulas:

- *Mean*: $E\left(X\right)=\sum_{s_i\in S}X\left(s_i\right)P\left(s_i\right)=\sum_i x_i p_i$ with $x_i = X(s_i)$, $p_i = P(s_i)$. Usually, the mean is denoted with μ.
- *Variance*: $Var\left(X\right)=E\left(\left(X-\mu\right)^2\right)=\sum_i\left(x_i-\mu\right)^2 p_i$.

Entropy

Let X be a random variable with corresponding values $x_1, \ldots, x_m$, and $p_i = P(X = x_i)$ for which the following equality occurs:

$$\sum_{i=1}^{n} p_i = 1$$

Let A be an event. In these settings, the entropy of the event A, which gives the uncertainty of A to occur, is computed using this formula:

$$H(A) = -\sum_{j=1}^{m} p_j \lg(p_j) = \sum_{j=1}^{m} p_j \lg\left(\frac{1}{p_j}\right)$$

When $p_i = 0$, the convention is as follows:

$$p_i \lg(p_i) = p_i \lg\left(\frac{1}{p_i}\right) = 0$$

When A, B are two events, the following relations take place:

Joint entropy

$$H(A,B) = \sum_{a,b} P(A = a, B = b) \lg\left(P(A = a, B = b)\right)$$

Conditional entropy

$$H(A|B = v) = -\sum_{m} P(A = m|B = v) \lg\left(P(A = m, B = b)\right)$$

Equivocation of A related to B

$$H(A|B) = \sum_{m} P(B = m) H(A|B = m)$$

In the previous relations, a, b go through all of the values within the random variables, A, B respectively, and m goes through all of the values within the random variable A.

A Little Algebra

Prime factorization. This is the process of decomposing a composite number into factors as prime numbers. The prime factorization is unique for each number, according to the fundamental theorem of arithmetic. Prime factorization is important in cryptography, because although it is easy (*easy* meaning a polynomial-time algorithm exists that does this) to check whether a number is prime, there is no general method through which the prime factorization can be achieved. For example, the prime factorization of 4320 is $4320 = 2^5 \bullet 3^3 \bullet 5^1$. Note here that 2, 3, and 5 are prime numbers. Prime factorization is one of the most important hardness assumptions in cryptography. One of the widely used cryptosystems, RSA, lies its hardness assumption on prime factorization.

BigInteger. Big integers are not quite mathematical tools. They are used in computer science (cryptography) to show that an integer value should have a very large value that exceeds the representation of an int. For example, in Java, there is a special class called `BigInteger`. Listing 7-1 is an example of using the `BigInteger` class in Java. Figure 7-1 shows the result.

Listing 7-1. *Using BigIntegers*

```
1   import java.math.BigInteger;
2
3   public class BigIntegerExamples {
4
5         public static void main(String[] args) {
6               BigInteger bi1, bi2;
7
8               //initialization with a string value that will be
                converted to BigInteger
9               bi1 = new BigInteger("5697726366552");
10              System.out.println(bi1);
11
12              //initialization with an int value
13              bi2 = BigInteger.valueOf(87362);
14              System.out.println(bi2);
15
```

```
16              //the operations are implemented as methods within
                the class
17              BigInteger sum = bi1.add(bi2);
18              System.out.println("Sum: " + sum);
19              BigInteger pow = bi1.pow(bi2.shortValue());
20              System.out.println("Power: " + pow);
21
22              //it provides complex operations such as modulo then power
23              BigInteger result = bi1.modPow(bi2, new BigInteger("5"));
24              System.out.println("Mod inverse: " + result);
25
26              //comparison between two BigIntegers
27              if(bi1.compareTo(bi2) < 0)
28                    System.out.println(bi1 + " less than " + bi2);
29              else
30                    System.out.println(bi1 + " greater than " + bi2);
31
32        }
33  }
```

Figure 7-1. *The result of using BigIntegers*

Be careful with the `.pow()` method on `BigInteger` because it can result in very large numbers, especially with large exponents. In real applications, consider the implications of performing such operations, as they can lead to performance issues or `OutOfMemoryError`.

The RSA problem. From the RSA encryption system, another hardness assumption resulted. The RSA problem is as follows: let (N, e) be the public key of the RSA cryptosystem and C be encrypted text (which is publicly available). It is required to compute efficiently P, where $C \equiv P^e (mod\ N)$. For more about the RSA cryptosystem, see Chapter 10.

Blum integer. A Blum integer is a number of the form $n = p \cdot q$, where p and q are prime numbers and are congruent to 3 mod 4. Listing 7-2 provides a way in which an integer value can be tested as a Blum integer or not. Figure 7-2 shows the result.

Listing 7-2. *Testing Whether a Number Is a Blum Integer*

```
import java.util.Scanner;

public class BlumInteger {
      public static boolean checkBlumInteger(int value)
    {
            //define an array with elements from 0 to value-1
            //suppose each number in this interval is prime
        int primesArray[] = new int[value + 1];
        for (int i = 0; i < value; i++)
              primesArray[i] = 1;

        //update the array with the primality of number within in the
        interval
        for (int i = 2; i * i <= value; i++) {
             if (primesArray[i] == 1) {
                   //if a value is prime, then certainly its multiples
                   are no primes
                   //update the primality accordingly
                for (int j = i * 2; j <= value; j += i)
                      primesArray[j] = 0;
            }
        }
```

```
21
22          //check whether the value is a product of two prime numbers
23          //on the same time check whether the value has a form of 4x+3
24          for (int aux = 2; aux <= value; aux++) {
25              if (primesArray[aux] == 1) {
26                  if ((value % aux == 0) && ((aux - 3) % 4) == 0) {
27                      int x = value / aux;
28                      return (x != aux && primesArray[x] == 1  &&
                        (x - 3) % 4 == 0);
29                  }
30              }
31          }
32          return false;
33      }
34
35      public static void main(String[] args)
36      {
37          int vals[] = {177, 125};
38          System.out.println("Checking Blum integer...");
39          for(int i=0;i<vals.length;i++)
40          {
41          if (checkBlumInteger(vals[i]) == true)
42              System.out.println(vals[i] + " - True");
43          else
44              System.out.println(vals[i] + " - False");
45          }
46      }
47  }
```

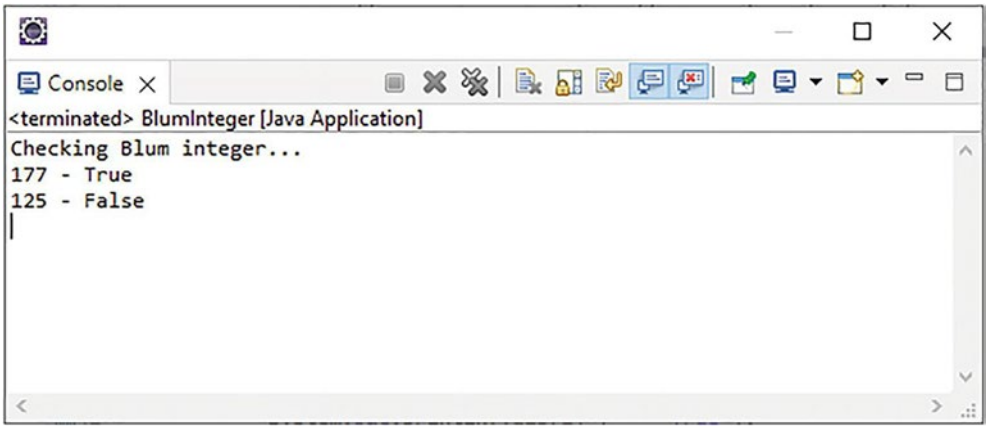


Figure 7-2. *The result of Listing 7-2*

Quadratic residue modulo *n*. This is pretty self-explanatory: it is an integer value q for which an integer value x exists, such that $x^2 \equiv q(mod\ n)$.

Algebraic structures. These are characterized by the following elements: A, which is a set having at least one element and a set of operations defined over the set A. The operations must meet some conditions. Examples of algebraic structures are groups, rings, and fields.

Groups. One of the most important elements in cryptography is the group. These are algebraic structures that meet some conditions. Let G be a set with at least one element and a binary operation denoted with + defined over the set G. In general, the groups we are working with are abelian. Therefore, the properties that a group has are as follows:

- + is well-defined over G. This means that $g_1 + g_2 \in G, \forall\ g_1, g_2 \in G$. However, usually, this property is satisfied by default, from the fact that it is a binary operation.
- + is associative. This means that $(g_1 + g_2) + g_3 = g_1 + (g_2 + g_3), \forall\ g_1, g_2, g_3 \in G$.
- Existence of an identity element (unique) $e \in G$ for which $g + e = e + g = g, \forall\ g \in G$.
- Existence of an inverse element (unique) $g' \in G$ for which $g + g' = g' + g = e$.
- If + is commutative, then the group is called abelian. + is commutative if $g_1 + g_2 = g_2 + g_1, \forall\ g_1, g_2 \in G$.

Another important algebraic structure used in cryptography is the *field*. A field is a tuple $(K, +, \bullet)$ for which (1)$(K, +)$ is an abelian group, (2)$(K, \bullet)$ is an abelian group, and (3)" • " is distributive over +, namely, $k_1 \bullet (k_2 + k_3) = k_1 \bullet k_2 + k_1 \bullet k_3, \quad \forall k_1, k_2, k_3 \in K$.

Let $x_1, \ldots, x_n \in G$, where G is a group. If for each $x \in G$ some integer values $\alpha_1, \ldots, \alpha_n$ exist such that

$$x = \alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_n x_n, \forall x \in G$$

then it is said that G is generated by $S = \{x_1, \ldots x_n\}$. A group that is generated by only one element is called a *cyclic group*. The *order of a group* is given by the number of its elements. The order of an element $g \in G$ represents the smallest integer m for which $g^m = e$, where $(G, \bullet)$ is the group.

Galois fields. Their name comes from the mathematician Evariste Galois and represents fields with a finite number of elements in the base set. Galois fields are important in computer science because here the data is represented as bits that represent the elements of the Galois field $GF = (Z_2 = \{0, 1\}, +, \bullet)$. Using field properties, different operations can be applied between bits; therefore, data can be scrambled easily. In the general case, a Galois field is defined as follows:

$$GF\left(p^n\right) = \{0,1,\ldots p-1\} \cup \{p,\ p+1,\ldots p+p-1\} \cup \left\{p^2,\ p^2+1,\ldots,\ p^2+p-1\right\} \cup \ldots$$
$$\cup \left\{p^{n-1},\ldots,\ p^{n-1}+p-1\right\}$$

In this form, p is a prime number, and n is a positive integer. The values p^n and p are special, as the first gives the order of the field, and the other gives the characteristic of the field. Note that addition and multiplication are made modulo p in a Galois field. The following is an example of *GF*:

$$GF\left(2^3\right) = \left\{0,1,2,2+1,2^2,2^2+1,2^2+2,2^2+2+1\right\} = \{0,1,2,3,4,5,6,7\}$$

An example of usage Galois fields in computer science is the ASCII code that contains 255 characters. The index of each character can be seen as an element of $GF(2^8)$.

Elliptic Curves

Elliptic curves have applications in different branches of cryptography, especially in digital signatures. These were introduced by Neal Koblitz [1] and Victor Miller [2] independently. Using elliptic curves brings advantages in the terms of efficiency because the size of the keys in an elliptic curves–based cryptosystem is less than other cryptosystems with the same level of security. Elliptic-curve cryptography (ECC) is a branch of public-key cryptography (PKC) based on the algebraic properties of the elliptic curves.

Weierstrauss form for elliptic curves. The graph that corresponds to the following equation is called an elliptic curve E:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

where a_1, a_2, a_3, a_4, a_6 are constants and $\Delta \neq 0$ (called the discriminant of E). The discriminant has the following form:

$$\Delta = -d_2^2 d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \text{ with}$$

$$d_2 = a_1^2 + 4a_2$$

$$d_4 = 2a_4 + a_1a_3$$

$$d_6 = a_3^2 + 4a_6$$

$$d_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4$$

The condition for delta to be different from 0 is required to eliminate the possibility of multiple roots. When defining an elliptic curve, it should be mentioned from which set the values a_1, a_2, a_3, a_4, a_6x, y are. In cryptography, usually these belong to Z_p (where p is a prime number) for which there are addition and multiplication operations, such that together they form a field structure. Another possible set for the values is Z_q, where $q = p^k$, $k \geq 1$, and p is prime. When the curve E has the values in a field K, then it is said that E is *defined over K*, and it is denoted with $E(K)$. However, the graph must contain a special point called the *point at the infinite* and noted with ∞, which represents the neutral element. Therefore, $E(K)$ has the following form:

$$E(K) = \{\infty\} \cup \{(x,y) \in K \times K | y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\}$$

For this structure, there is an operation called *point addition*, for which the neutral element is ∞, together forming a group structure $(E(K), +)$. Elliptic-curve cryptography (ECC) lies its hardness assumption on the *discrete logarithm problem (DLP) for elliptic curves*, which states the following:

Given $G = (E(Z_p), +)$, $P \in E(Z_p)$, $Q \in H = \{sP, s \geq 0\}$, compute the unique integer value k, such that $Q = kP$, where $kP = \underbrace{P + \ldots + P}_{k \text{ times}}$.

Note that the operation kP represents the multiplication of a point on the curve with a scalar value. You can find more about elliptic curves arithmetic and its use in cryptography in [3]. Note that DLP can be generalized to any group structure.

One of the most efficient attacks against elliptic curve cryptosystems is the *Pollard-Rho attack* that speculates on the discrete logarithm problem for elliptic curves.

First, it is given the general case of the DLP: let p be a prime number, a a generator for the Diffie-Hellmann problem, x the private key, and y the corresponding public key. Between all of these, the following equality exists:

$$y \equiv a^x \bmod p$$

The general case of DLP requires us to compute x given y, p, a.

The Pollard-Rho attack is a statistical algorithm in which the bit security is divided by two; therefore, it is more efficient than brute-force attacks. While brute-force attacks require n trials, the Pollard-Rho attack requires only $\sqrt{n}$ trials. Its purpose is to find the values $u, v, u', v' \in \{0, \ldots, p-2\}$ for which $a^u y^v = a^{u'} y^{v'}$. From this equality and the equality from earlier (DLP), it results in $a^{u+xv} = a^{u'+xv'}$; therefore, $u + xv = u' + xv' \ mod\ (p-1)$. A solution for this equation (i.e., finding x) is found when the equation is solved within the modular ring modulus $p - 1$. Here, one of the following things can happen:

- The solution does not exist. In this situation, other values for u, v, u', v' should be found.
- The solution exists, but it is not unique. In this situation, if there is a relatively small number of solutions, then all of them can be tested by brute force.

Test Suites

Ensure the security and effectiveness of cryptographic algorithms and their implementations. Various test suites are designed to validate these aspects based on internationally established standards.

The *National Institute of Standards and Technology* (NIST) has an important role in the advancement and validation of cryptographic standards through its various test suites. These suites are designed to rigorously validate the security and reliability of cryptographic algorithms and modules, ensuring they meet the federal guidelines established by international standards. Among these, the *Cryptographic Algorithm Validation Program* (CAVP) is used for testing cryptographic algorithms for compliance with NIST standards, covering a broad spectrum from symmetric encryption to complex public key algorithms like RSA and ECC. Another significant suite, the *Cryptographic Module Validation Program* (CMVP), certifies cryptographic modules to *Federal Information Processing Standards* (FIPS), which are critical for government and military communications. Additionally, NIST's involvement in transitioning to Post-Quantum Cryptography highlights its forward-thinking approach, aiming to standardize cryptographic algorithms resistant to quantum computing threats. Through these test suites, NIST ensures that cryptographic systems protect sensitive information against evolving threats, maintaining integrity, confidentiality, and trust in digital communications and data storage.

The following are some of the most important NIST test suites:

- **Cryptographic Algorithm Validation Program (CAVP) [12].** Validates cryptographic algorithms against NIST standards, covering a wide range of cryptographic operations, including those based on RSA, ECC, and algorithms where prime factorization plays a critical role.
- **Transitioning to Post-Quantum Cryptography [13].** Focuses on standardizing cryptographic algorithms that are secure against quantum computing attacks. This involves evaluating hardness assumptions and cryptographic primitives for their quantum resistance.

- **NIST Randomness Tests (Special Publication 800-22) [14].** A suite designed to test the randomness of binary sequences produced by cryptographic random number generators, relevant to the secure generation of pseudorandom numbers, including uses of Blum integers.

The following are other important test suites:

- **SECG Test Suites for ECC Validation [15].** Defined by the Standards for Efficient Cryptography Group (SECG), focuses on the implementation of Elliptic Curve Cryptography, ensuring compliance with the standards for elliptic curve-based cryptosystems.
- **RSA Laboratories' PKCS#1 (PKCS #1 v2.2 (RFC 8017)) [16].** Specifies standards for RSA encryption and signature schemes, testing the secure implementation of RSA-related cryptographic operations.
- **Cryptographic Module Validation Program (CMVP) [17].** Validates modules to Federal Information Processing Standards (FIPS 140-2, transitioning to FIPS 140-3). It is a joint effort by NIST and the Canadian Centre for Cyber Security that certifies cryptographic modules, including those implementing the mathematical foundations of cryptographic protocols discussed.

Conclusion

This chapter presented beginner background information (definitions and mathematical descriptions) for the world of cryptography. You can find detailed references for mathematical concepts that represent the background of cryptography in any of the following references.

References

[1] Koblitz, N. (1987). Elliptic curve cryptosystems. Mathematics of computation, 48(177), pp. 203–209.

[2] Miller, V. S. (1985). Use of elliptic curves in cryptography. In Conference on the theory and application of cryptographic techniques (pp. 417–426). Springer, Berlin, Heidelberg.

[3] Hankerson, D., Menezes, A. J., Vanstone, S. (2006). Guide to elliptic curve cryptography. Springer Science Business Media.

[4] Gilbert, L. (2014). Elements of modern algebra. Nelson Education.

[5] Dent, A. W. (2006). Fundamental problems in provable security and cryptography. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 364(1849), pp. 3215–3230.

[6] Stinson, D. R., Paterson, M. (2018). Cryptography: theory and practice. CRC Press.

[7] Oppliger, R. (2011). Contemporary cryptography. Artech House.

[8] Buchmann, J. (2004). Introduction to cryptography (Vol. 335). New York: Springer.

[9] Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A. (2018). Handbook of applied cryptography. CRC Press.

[10] Etienne, E. (2019). Elementary statistical methods of cryptography.

[11] Mihailescu, M. I., & Nita, S. L. (2021). Pro Cryptography and Cryptanalysis with C++ 20: Creating and Programming Advanced Algorithms. Apress.

[12] Cryptographic Algorithm Validation Program. [Online] Available: https://www.nist.gov/programs-projects/cryptographic-algorithm-validation-program

[13] Migration to Post-Quantum Cryptography. [Online] Available: https://www.nccoe.nist.gov/crypto-agility-considerations-migrating-post-quantum-cryptographic-algorithms

[14] A. Rukhin et al., A statistical test suite for random and pseudorandom number generators for cryptographic applications, vol. 22. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2001.

[15] D. R. L. Brown, "SEC 1: Elliptic Curve Cryptography." [Online] Available: https://www.secg.org/sec1-v2.pdf

[16] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2," Internet Engineering Task Force, Request for Comments RFC 8017, Nov. 2016.

[17] I. T. L. Computer Security Division, "Cryptographic Module Validation Program | CSRC | CSRC," CSRC | NIST. [Online]. Available: https://csrc.nist.gov/projects/cryptographic-module-validation-program

CHAPTER 8

Pseudorandom Number Generators

Randomness is a key element in cryptography, as cryptosystems and protocols are based on arbitrary numbers that cannot be predicted. Numbers with such characteristics are known as *randomly generated numbers* (or *random numbers*), and the technique of their generation lies in a statistical context. Random numbers are mainly used in domains in which results that cannot be predicted are expected, for example, statistical sampling, simulations, or even gambling.

There are two main ways in which random numbers can be generated. The first method is based on phenomena from physics that prove to be random, generating great entropy values, and is called *true random number generators* (TRNGs). The following are examples of TRNGs:

- **Quantum random number generators.** Use quantum phenomena, such as the unpredictability of quantum decay, to generate true randomness. They are considered the gold standard in randomness due to the inherent unpredictability of quantum processes.
- **Atmospheric noise.** Randomness can be generated from natural atmospheric noise or other physical processes like thermal noise. Websites and services might use radio receivers or other sensors to capture this data and convert it into random numbers.
- **Hardware random number generators.** These are devices or computer peripherals that generate random numbers from physical processes, such as electronic noise in circuits. They often serve as entropy sources for *cryptographically secure pseudorandom number generators* (CSPRNGs) in cryptographic applications.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_8

The second way is to create algorithms that can "choose"/compute a sequence of numbers that appears to be random. Such algorithms are called *pseudorandom number generators* (PRNGs), and usually, their base is a special value called a *seed* from which the random generation of numbers starts in a specific way. Because the PRNGs use this initial value seed, they are not truly random, which is the reason for the *pseudo* in their name. However, the seed may come from natural sources and may be changed occasionally. Another way to generate pseudorandom numbers consists of combining the two main approaches.

Statistically, there is not too much information about what "random bits" refers to, but from the computing machine point of view, *random bits* refers to a sequence of bits chosen randomly.

To be considered a PRNG, an algorithm should meet the following conditions:

- It is characterized by simplicity and fastness.
- The sequence of numbers that it generates should have an arbitrary length, and the numbers should not repeat. Limited by the technology, a computer (or another device) can generate numbers only smaller than a value, even if that value is the maximum (highest) number that the computer supports; therefore, the period cannot be infinite. However, the period can be set as high as possible.
- The numbers should not depend on each other.
- The generated numbers should be uniformly distributed from the statistical point of view.

To be used in cryptography, a PRNG should have the following additional properties (these are called CSPRNG):

- **Next-bit test [1].** Fixing an arbitrary integer k, an attacker cannot predict the value of the bit positioned on the $(k + 1)$ index of the sequence using a polynomial-time algorithm if it knows the values of the first k bits but does not know the seed. In [1], the author proved that a PRNG passing the next-bit test will pass all statistical tests for randomness in polynomial time.
- **State compromise extensions.** Breaking a subsequence of the pseudorandom sequence of numbers, an attacker cannot recover the entire sequence.

In statistics, there is still some information about random bits: the frequency of the appearance of 1 and 0 should be the same; the frequency for 00, 11, 01, 10 should be the same, but these should appear half as less often than 1 and 0. Different statistical tests may be used to test "how much" a sequence of numbers is random, such as Diehard [2] or Kolmogorov-Smirnov [3] tests. Randomness is an essential aspect used in cryptography, and deducing or recovering a random number should be very difficult (ideally impossible). To be considered a perfect random number, the number may be recovered by an attacker only by using brute force. Many techniques from cryptanalysis exploit the weaknesses of the functions that generate random numbers.

Depending on the purpose, the required randomness degree of the PRNG can be different. For example, when keys of an encryption system are generated, the degree should be very high, while in protocols uniqueness for the generated nonce is required, and the degree of randomness is not required to be very high.

The following are some standards that describe the PRNGs:

- **FIPS 186-4 [4].** Published by the U.S. National Institute of Standards and Technology (NIST), this standard specifies digital signature algorithms suitable for applications requiring data integrity. It includes guidance on using secure random number generators during the generation of digital signatures. *Relevance to PRNGs*: Describes acceptable methods for generating random bits needed for cryptographic algorithms, ensuring the security and integrity of cryptographic keys.
- **NIST SP 800-90A [5].** This special publication provides recommendations for the construction of CSPRNGs. It is part of a series aimed at ensuring the security and quality of random number generation. *Relevance to PRNGs*: Specifically details methods for generating cryptographically secure random numbers, including approved deterministic random bit generators (DRBGs).
- **ANSI X9.18-1985 Appendix C [6].** This standard, developed by the American National Standards Institute (ANSI), focuses on financial services. Appendix C deals specifically with methods for generating random numbers for cryptographic keys. *Relevance to PRNGs*: Provides early guidelines for generating random numbers in financial applications, emphasizing the security requirements for financial transactions.

- **ANSI X9.31-1998 Appendix A.2.4 [6].** Another ANSI standard tailored for the financial industry, this one outlines algorithms for digital signatures and key management. *Relevance to PRNGs*: Appendix A.2.4 describes a method for generating pseudorandom numbers to be used in cryptographic applications, ensuring that keys and other sensitive data are protected.
- **ANSI X9.62-1998 Annex A.4, which is obsoleted by ANSI X9.62-2005, Annex D (HMAC_DRBG) [6].** Initially focused on elliptic curve cryptography (ECC), providing standards for ECC public key cryptography, the Annex A.4 included specifications for generating pseudorandom numbers in the context of ECC. The 2005 update introduced Annex D, which obsoleted the earlier version's annex related to PRNGs. *Relevance to PRNGs* (2005 update): Annex D introduces HMAC_DRBG, a deterministic random bit generator based on the HMAC algorithm. This method is known for its security and efficiency in generating cryptographically secure pseudorandom numbers.

Examples of PRNGs

In this section, we present some PRNGs that are not necessarily as important as PRNGs themselves but are important from the technical point of view and the mechanisms used, because they represent the basis for other more complex or powerful PRNGs.

Linear Congruential PRNGs

The linear congruential generator (LCG) is a type of PRNG notable for its simplicity and ease of implementation. It is one of the most widely used methods for generating sequences of numbers that approximate the properties of random numbers. In [7], the author introduced a simple linear congruential generator, given by the following recurrence:

$$r_{n+1} = ar_n + b \ (mod \ m),$$

with $a, b, m \in \mathrm{N}$ constants and r_0 being the seed. A general formula for linear congruential PRNGs is as follows:

$$r_{n+k} = \left(a^k r_n + \left(a^k - 1\right)cb^{-1}\right)\left(mod\ m\right)$$

In these types of PRNGs, cycles occur, with the maximum length (called a *period*) of cycles being m. The advantage of linear congruential PRNGs is the fastness of computations. Its general form achieves the maximum period in the following situations:

- $\gcd(c, m) = 1$
- $b = a - 1$ is multiple of a prime number that divides m
- $b \ mod\ 4 = 0$ if $m \ mod\ 4 = 0$

LCG's advantages include simplicity (the algorithm is straightforward to implement in software), speed (it requires minimal computational resources, making it very fast), and predictability (given the seed, the sequence can be reproduced, useful for debugging and testing purposes).

Linear congruential PRNGs cannot be used in cryptography because they are very vulnerable: if an attacker can discover the seed, then it breaks the generator. Disadvantages of LCG from cryptography's perspective include periodicity (the sequence inevitably repeats, which may be undesirable for certain applications), nonrandomness (statistical properties of the sequence can be inferior to more complex methods, making it unsuitable for cryptographic purposes or high-quality simulations), and predictability (if an attacker knows the algorithm's parameters and part of the sequence, they can predict future values, compromising security in cryptographic applications). Comprehensive cryptanalysis is presented in [8].

Blum-Blum-Shub PRNG

A simple and efficient CSPRNG is Blum-Blum-Shub (BBS), also called a *residual generator* [9], based on Blum integers (see Chapter 7 for a description of a Blum integer). To generate random numbers using BBS, one should proceed as follows [9]:

- Establish an arbitrary integer k, choose two prime numbers p, q represented on $k/2$ bits, and let $n = pq$.

- Compute x_0 as a quadratic residual modulo n and define the series $r_{i+1} = r_i^2 \left(mod\ n \right)$.
- Set $s_i = r_i (mod\ 2)$, $1 \le i \le m$ and generate the number $f(x_0) = s_1 s_2 \ldots s_m$, where m is the length of the series at an arbitrary moment.

As s_i can be determined independently as $s_i = x_0^{2^i (mod\ (p-1)(q-1))}\ mod\ 2$, the BBS PRNG does not lay into recursion. BBS is constructed on the hard problem of factoring integers, in this case, factoring n. Integer n can be public, as there is no restriction for who may generate pseudorandom numbers, but n should be difficult to be factored (ideally impossible), in which case the value that is generated cannot be predicted. Another important feature of BBS is that if an attacker obtains a subsequence of numbers generated by BBS PRNG, then it cannot predict the previous bit or the next bit. Although it is a good PRNG, BBS is inefficient. A more efficient variation of it is presented in [10], in which considering l the length of x_i, and then the last $\lfloor \log_2 x_i \rfloor$ are used further. In cryptography, BBS is one of the best PRNGs used in protocols for generating and distributing the keys involved in a cryptosystem.

Linear Circuit PRNGs

Error detection and correction theory and linear cryptosystems (for example, AES) use linear circuits (also called *shift registers*) because the computations are fast. In short, the linear circuit, called a *linear feedback shift register* (LFSR), is a serial register with a feedback function. LFSRs are widely used in fields ranging from cryptography to digital signal processing due to their simplicity and efficiency in generating sequences of pseudorandom bits. LFSRs utilize linear feedback mechanisms to produce long sequences of bits from a short initial value (the seed). The *linear* in their name refers to the use of linear functions in their feedback loops, which are usually implemented with exclusive-or (XOR) operations. An LFSR consists of a shift register and a feedback function. The shift register is a sequence of bits that shifts one position at each step, discarding the last bit and introducing a new bit at the start, determined by the feedback function. The feedback function is typically a linear combination of certain bits in the register (called *taps*), combined using XOR operations. The choice of taps determines the properties of the sequence, such as its maximum length and statistical randomness. An n–LFSR is a register with n flip-flops of data (DF-F). You can find more information about shift registers in [11].

A Geffe generator [12] is a PRNG that uses three LFSRs to combine their nonlinearity. However, a Geffe generator is not resistant against a correlation attack, because the number generated by the generator is the same as the output of the second LFSR in 75 percent of cases. Knowing the polynomial representation of the LFRSs, it would be easy to compute the seed and the series produced by the second LFRS and then to compute the frequency of when the result produced by the second LFSR is the same as the result produced by the PRNG. A similar deduction can be made for the third LFSR, for which the Geffe PRNG also produces similar results in 75 percent of cases. Correlating these two pieces of information, then the entire sequence of generated numbers can be recovered [12].

Other types of LFSR generators are called *stop-and-go*. Some examples of stop-and-go PRNGs are the Beth-Piper PRNG [13] (based on three circuits for which the clocks are controlled; however, it does not resist against correlation attack) and the Gollmann PRNG [14] (based on a serial construction of LFSRs, in which the clock of the current LFRS is controlled by the previous LFSR).

Other PRNGs

The PRNGs lie their difficulty of recovering the sequence or predicting the next value in different mathematical hardness assumptions.

An example of such a mathematical problem is the discrete logarithm problem that represents the base for the Blum-Micali PRNG [15]. The setup for it consists of p, g, which are prime numbers, and an initial value of x_0. A sequence of random numbers is given by the formula $x_{i+1} = g^{x_i} \pmod{p}$. If $x_i < \frac{p-1}{2}$, then the exit is 1; otherwise, the exit is 0. For a large prime number p, the Blum-Micali PRNG is secure.

Another example of a mathematical problem that can be used in PRNGs is the RSA problem [16], which represents the hardness of breaking the RSA cryptosystem. The initial setup for this PRNG consists of the following values: p, q for large prime numbers, $n = pq$; the value e that accomplishes the following criteria $gcd(e, (p-1)(q-1)) = 1$; and an initial value $x_0 < n$. A sequence of random numbers is given by the formula $x_{i+1} = x_i^e \pmod{n}$. The exit of the RSA PRNG is $z_i = x_1 (mod\ 2)$. For a large number n, the generator is secure.

Another type of PRNG is the Ranrot class [17] used in Monte Carlo algorithms, which use Fibonacci numbers in their generation along with a shifting operation. Mainly, there are four types of Ranrot generators.

PRNGs Security

As you have already seen, pseudorandom number generators are extremely important in cryptography. Because these are used in many cryptographic techniques, the PRNGs themselves should be secure. [18] is a good reference for attacks applied to PRNGS, as the author explores a wide range of attacks, grouping them into these three categories:

- **Direct cryptanalytic attacks.** In these types of attacks, the attacker can distinguish between the output values of the PRGN and the random outputs. This category of attacks can be applied to almost all use cases of PRNGs.
- **Input-based attacks.** In these types of attacks, the attacker can use information about the input values or control them in order to find the vulnerabilities of the PRNG. This also leads to distinguishing between the output values of the PRGN and the random outputs.
- **State compromise extension attacks.** In these types of attacks, the attacker can use further information about the internal state of the PRNG. These attacks can be launched in situations in which at a certain time the internal state of the PRNG was compromised (even for a short time) and the attacker is able to collect further information or take further advantages, although the situation was fixed.

A detailed description of these classes of attacks can be found in [18], which also presents examples of attacks that can be applied to known PRNGs. A more recent work about PRNGs security is [19].

Java.util.Random Class

In Java, random numbers can be generated using the `java.util.random.RandomGenerator` interface [20], which can be instantiated using one of the following constructors. To create the `randomObject`, it was called the `getDefault()` method that returns a `RandomGenerator` that meets the minimum criteria of having an algorithm with a state size of 64 bits or more [20].

Using this interface, you can generate streams of `int`, `double`, and `long` random numbers or `int`, `double`, `float`, `long`, and `boolean` values. The seed is represented in 48 bits but then changed through a linear congruential relationship. Note that the

individual values are generated using a uniform distribution, which means any value has (almost) the same chance of being generated as the rest of the values of the same type. See Listing 8-1.

Listing 8-1. Using Java.util.Random

```
1   import java.util.random.RandomGenerator;
2
3   public class RandomExamples {
4
5       public static void main(String[] args) {
6
7           RandomGenerator randomObject = RandomGenerator.getDefault();
8
9           System.out.println("Generating an integer value < 1000... " +
            randomObject.nextInt(1000));
10          System.out.println("Generating an integer value... " +
            randomObject.nextInt());
11          System.out.println("Generating a long value... " +
            randomObject.nextLong());
12          System.out.println("Generating a Boolean value... " +
            randomObject.nextBoolean());
13          System.out.println("Generating a double value... " +
            randomObject.nextDouble());
14          System.out.println("Generating a float value... " +
            randomObject.nextFloat());
15
16          byte[] byteArray = new byte[15];
17          randomObject.nextBytes(byteArray);
18          System.out.println("Generating a sequence of bytes...");
19
20          System.out.println(java.util.Arrays.toString(byteArray));
21      }
22  }
```

The code from Listing 8-1 is straightforward, and it generates different types of random values. Figure 8-1 shows the output.

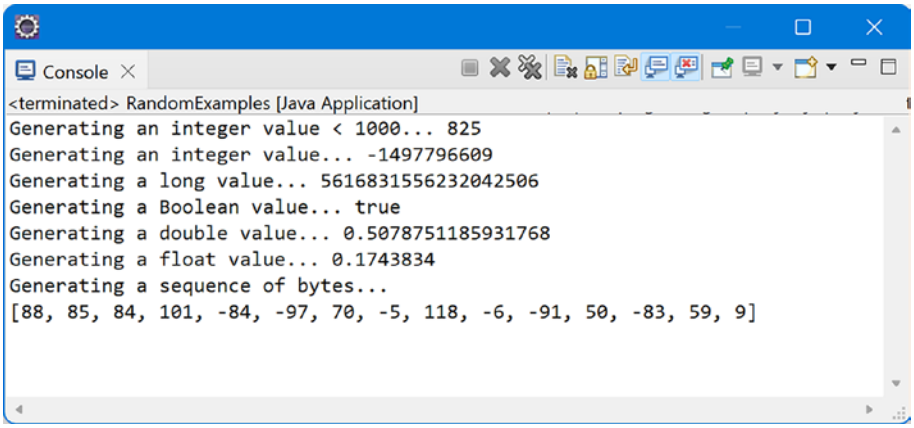

Console
<terminated> RandomExamples [Java Application]
Generating an integer value < 1000... 825
Generating an integer value... -1497796609
Generating a long value... 5616831556232042506
Generating a Boolean value... true
Generating a double value... 0.5078751185931768
Generating a float value... 0.1743834
Generating a sequence of bytes...
[88, 85, 84, 101, -84, -97, 70, -5, 118, -6, -91, 50, -83, 59, 9]

Figure 8-1. *Output for Listing 8-1*

However, the instances of `Random` are not cryptographically secure. To use secure PRNGs, Java provides a special class called `java.security.SecureRandom` [21]. The PRNGs in this class are implemented according to FIPS 140-2 and RFC 1740 standards. The ways in which the objects of the `SecureRandom` class are used are similar to how `Random` is used. In Listing 8-2 we adapted the previous example to generate securely random values. Figure 8-2 shows the output.

Listing 8-2. Generating Securely Different Types of Values

```
1   import java.security.SecureRandom;
2
3   public class SecureRandomExample {
4
5           public static void main(String[] args) {
6           SecureRandom secureRandomObject = new SecureRandom();
7
8           System.out.println("Securely generating an integer value <
            1000... " + secureRandomObject.nextInt(1000));
9           System.out.println("Securely generating an integer value... "
            + secureRandomObject.nextInt());
10          System.out.println("Securely generating a long value... " +
            secureRandomObject.nextLong());
11          System.out.println("Securely generating a Boolean value... " +
            secureRandomObject.nextBoolean());
```

```
12          System.out.println("Securely generating a double value... " +
            secureRandomObject.nextDouble());
13          System.out.println("Securely generating a float value... " +
            secureRandomObject.nextFloat());
14          System.out.println("Securely generating an integer number... "
            + secureRandomObject.nextGaussian());
15
16          byte[] byteArray = new byte[15];
17          secureRandomObject.nextBytes(byteArray);
18          System.out.println("Securely generating a sequence of
            bytes...");
19          System.out.print("[");
20          for(int i = 0; i< byteArray.length; i++)
21          {
22              System.out.print(byteArray[i] + " ");
23          }
24          System.out.print("]\n");
25      }
26  }
```

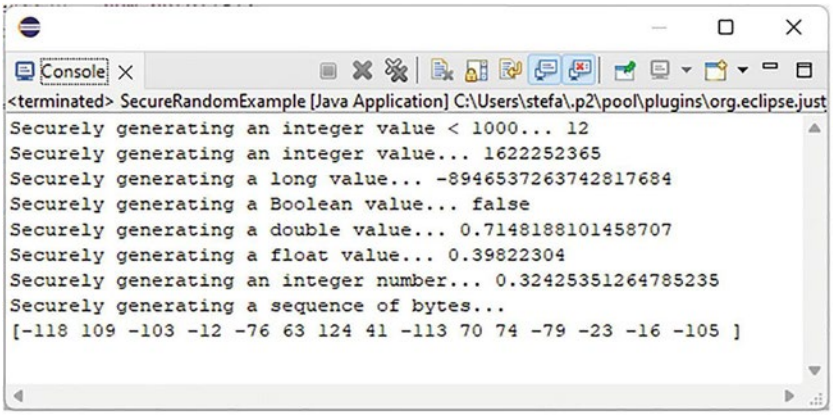

Console
<terminated> SecureRandomExample [Java Application] C:\Users\stefa\.p2\pool\plugins\org.eclipse.just
Securely generating an integer value < 1000... 12
Securely generating an integer value... 1622252365
Securely generating a long value... -8946537263742817684
Securely generating a Boolean value... false
Securely generating a double value... 0.7148188101458707
Securely generating a float value... 0.39822304
Securely generating an integer number... 0.32425351264785235
Securely generating a sequence of bytes...
[-118 109 -103 -12 -76 63 124 41 -113 70 74 -79 -23 -16 -105]

Figure 8-2. *Output for Listing 8-2*

The fundamental difference between `java.util.Random` and `java.security.SecureRandom` is given by how the seed is chosen. In `java.util.Random` the seed is generated based on the clock of the system, while in `java.security.SecureRandom` the seed is generated based on random data from the operating system.

Further, several secure coding practices are recommended when working with PRNGs.

- Use PRNGs already implemented, tested, widely used, and standardized. In this way, most of the possible vulnerabilities are fixed, and a high degree of security is ensured.
- Use PRNGs with strong security in applications; use, for example, `java.security.SecureRandom` instead of `java.util.Random`. However, the goal for which PRNG is used is very important. There are situations in which the applications need faster PRNGs rather than strong, secure PRNGs. On the other hand, other applications will require strong secure PRNGs rather than efficiency.
- Use a set of generated values only one time. Each time a series of pseudorandom numbers is needed, the best approach is to generate a new sequence.
- The values generated by the PRNG should be secured. If they should be stored somewhere, in a database or on a server, make sure that the database or the file that contains the generated values are strongly secured.

Example of Using CSPRNGs

An example of a scenario where CSPRNGs can be used is the authentication process in a web-based application. This application allows users to register, log in, and perform secure transactions. A common feature in such systems is the generation of one-time passwords (OTPs) for two-factor authentication (2FA) or the creation of session identifiers (session IDs) to manage user sessions securely. For both OTPs and session IDs, it is important that these values are unpredictable and cannot be easily guessed or regenerated by attackers. This is where secure random numbers come into play.

When a user opts for 2FA, the system generates an OTP to be sent to the user's registered email or phone number. The OTP can be generated using `SecureRandom` to ensure that it is cryptographically strong and hard to predict, as follows:

```
var secureRandom = new SecureRandom();
var otp = secureRandom.nextInt(1000000); // Generates a number between 0
                                            and 999999
System.out.println(String.format("%06d", otp)); // Ensures the OTP is
                                                   always 6 digits
```

The OTP is then hashed and stored temporarily in the database, with a timestamp to ensure it expires after a short duration, enhancing security. Upon successfully logging in, the user is assigned a session ID that is used to track the session's state across multiple requests. This ID must be secure to prevent session hijacking.

```
var secureRandom = new SecureRandom();
byte[] sessionBytes = new byte[24]; // 192 bits
secureRandom.nextBytes(sessionBytes);
var sessionId = java.util.Base64.getUrlEncoder().withoutPadding().
encodeToString(sessionBytes);
System.out.println(sessionId);
```

The session ID is then sent to the user's browser as a secure cookie. On the server side, the session ID is mapped to the user's session data, ensuring that user actions can be authenticated and authorized securely.

Using `SecureRandom` in these scenarios is important because it generates cryptographically strong random numbers that are computationally infeasible to predict. In security-sensitive applications, such as user authentication systems, using cryptographically secure random numbers helps protect against attacks that exploit predictable randomness, such as session hijacking or OTP guessing attacks.

Conclusion

In this chapter, we discussed pseudorandom number generators. You learned which characteristics an algorithm needs to have to be considered a pseudorandom generator. In addition, to be used in cryptography, the pseudorandom generator must have two additional characteristics. Further, we presented some PRNGs and the technical aspects behind them, and lastly, we provided straightforward examples of how `java.util.random.RandomGenerator` and `java.security.SecureRandom` can be used.

References

[1] Yao, A. C. (1982, November). Theory and application of trapdoor functions. In 23rd Annual Symposium on Foundations of Computer Science (SFCS 1982) (pp. 80–91). IEEE.

[2] Marsaglia, G. (1996). DIEHARD: a battery of tests of randomness. Available online: http://stat.fsu.edu/geo.

[3] Massey Jr, F. J. (1951). The Kolmogorov-Smirnov test for goodness of fit. Journal of the American Statistical Association, 46(253), 68–78.

[4] FIPS PUB 186-4, Available online: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[5] NIST 800-90A Rev. 1, Available online: https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final.

[6] Random Bit Generation, Available online: https://csrc.nist.gov/projects/random-bit-generation.

[7] Lehmer, D. H. (1951). Mathematical methods in large-scale computing units. Annu. Comput. Lab. Harvard Univ., 26, 141–146.

[8] Plumstead, J. B. (1982, November). Inferring a sequence generated by a linear congruence. In 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982) (pp. 153–159). IEEE.

[9] Blum, L., Blum, M., & Shub, M. (1986). A simple unpredictable pseudo-random number generator. SIAM Journal on computing, 15(2), 364–383.

[10] Sidorenko, A., & Schoenmakers, B. (2005, December). Concrete security of the Blum-Blum-Shub pseudorandom generator. In IMA International Conference on Cryptography and Coding (pp. 355-375). Springer, Berlin, Heidelberg.

[11] Goresky, M., & Klapper, A. (2012). Algebraic shift register sequences. Cambridge University Press.

[12] Wei, S. (2006). On generalization of Geffe's generator. IJCSNS International Journal of Computer Science and Network Security, 6(8A), 161–165.

[13] Beth, T., & Piper, F. (1984, April). The Stop-and-Go Generator. In Eurocrypt (Vol. 84, pp. 88–92).

[14] Park, S. J., Lee, S. J., & Goh, S. C. (1995, August). On the security of the Gollmann cascades. In Annual International Cryptology Conference (pp. 148-156). Springer, Berlin, Heidelberg.

[15] Blum, M., & Micali, S. (1984). How to generate cryptographically strong sequences of pseudorandom bits. SIAM journal on Computing, 13(4), 850–864.

[16] Friedlander, J. B., Lieman, D., & Shparlinski, I. E. (1999). On the distribution of the RSA generator. In Sequences and their applications (pp. 205–212). Springer, London.

[17] Fog, A. (2001). Chaotic random number generators with random cycle lengths. Available online: http://www.agner.org/random/theory/chaosran.pdf

[18] Kelsey, J., Schneier, B., Wagner, D., & Hall, C. (1998, March). Cryptanalytic attacks on pseudorandom number generators. In International workshop on fast software encryption (pp. 168–188). Springer, Berlin, Heidelberg.

[19] Ruhault, S. (2017). SoK: Security models for pseudo-random number generators. IACR Transactions on Symmetric Cryptology, 506–544.

[20] Class Random, https://docs.oracle.com/en%2Fjava%2Fjavase%2F21%2Fdocs%2Fapi%2F%2F/java.base/java/util/random/RandomGenerator.html.

[21] Class SecureRandom, https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html.

CHAPTER 9

Hash Functions

In information security, hash functions are an important cryptographic tool. They have applications in a wide range of use cases, such as securing passwords in a database, securing authentication, and maintaining message integrity.

In a few words, a *hash function* is a compression mathematical function that takes an input with any length and converts it into a string with a fixed length. The input can be almost anything such as text or different types of files. The output, on the other hand, should be unique for every input and has a predefined fixed length according to the type of the hash function. The output is called a *message digest*, *hash value*, or *hash*.

A hashing class is a tuple with four elements, as follows [1]: the set of all possible messages (encrypted or not) X; a finite set of fingerprints (digests) Y; a finite set of keys K; and a set H of hash functions such that for every $k \in K$ there exists a hash function $h_K \in H$, $h_K : X \to Y$. A pair $(x, y) \in X \times Y$ is valid for the key K if $h_K(x) = y$.

The following problems should be hard to compute [1]:

- **Noninvertible.** For a given $y \in Y$, it is hard to discover $x \in X$, such that $y = h(x)$. This property means that it is hard to invert the function h, and it ensures that if an attacker has an output value, then it cannot compute the input value.
- **Weak collisions.** For a given pair (x, y), it is hard to discover $x_1 \neq x$ such that $h(x_1) = h(x)$. This property protects against situations in which the attacker has the input value and the corresponding hash and tries to modify legitimate values with other values that seem to be legitimate.
- **Collisions.** It is hard to find two different values $x, x_1 \in X$ such that $h(x) = h(x_1)$.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_9

By "hard" we mean that solving the problem involves only algorithms of high complexity (eventually, the problem is NP-complete). A hash function $h : X \to Y$ for which the collision problem is difficult is called a *collision-resistant* hash function. Because $\#X > \# Y$, any hash function admits collisions, but the trick is to make it extremely difficult to find the collisions and only by using algorithms with high complexity. (We denoted with $\#A$ the cardinal of the set A, namely, the number of elements contained by the set.)

It is enough that a hash function is collision-resistant because this property implies weak collision and noninvertibility.

Figure 9-1 shows an example of a hash function. The plain message is "Sample Plaintext," and after applying the hash function SHA-512 to it, the text on the right is obtained.

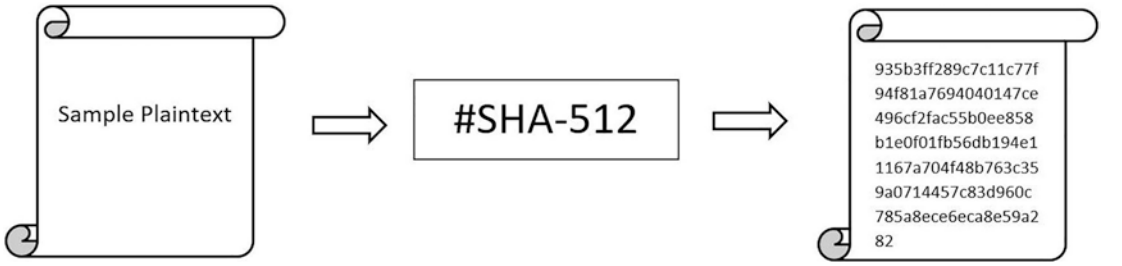


Figure 9-1. *Applying SHA-512 to text*

Several theorems establish an inferior limit for the cardinal of Y. One of them is the birthday paradox, through which a relation between $\#X$ and $\#Y$ is determined. The relation is as follows: $\#X \geq 2 \bullet \# Y$. The birthday attack is a brute-force attack that follows to discover collisions. Generally, an attacker randomly generates some messages x_1, x_2, ... $\in X$ (in practice, they use pseudorandom number generators) and, for each x_i computes and stores the digest $y_i = h(x_i)$. Then, the attacker compares y_i with the ones stored previously. If y_i equals another stored y_j, then the attacker found a collision whose form is (x_i, x_j). According to the birthday paradox, this would happen after about $2^{\frac{N}{2}}$ messages. A hash function that represents the hash value on 40 bits is vulnerable to the birthday attack using only 2^{20} (approximately one million) random messages. Therefore, in practice, it is recommended to use hash functions that represent the values on at least 256 bits. Table 9-1 presents a list of keyed cryptographic hash functions, while Table 9-2 presents a list of unkeyed cryptographic functions [2].

Table 9-1. *Keyed Cryptographic Functions*

Name	Length of the Tag (Bits)	Type	References
BLAKE2	Arbitrary	Keyed hash function with prefix-MAC	[3], [4]
BLAKE3	Arbitrary	Keyed hash function with supplied initializing vector (IV)	[5]
HMAC	-	-	[6]
KMAC	Arbitrary	Based on Keccak	[7], [8]
MD6	512	Merkle tree with NLFSR	[9]
PMAC	-	-	[10]
UMAC	-	-	[11]

Table 9-2. *Unkeyed Cryptographic Functions*

Name	Length of the Tag (Bits)	Type	References
BLAKE256	256	HAIFA structure [12]	[15]
BLAKE512	512	HAIFA structure [12]	[15]
gost	256	Hash	[16]
md2	128	Hash	
md4	128	Hash	[17]
md5	128	Merkle-Damgard construction [13]	[18]
md6	Up to 512	Merkle tree NLFSR	[9]
ripemd	128	Hash	[19], [21]
ripemd-128	128	Hash	[19]
ripemd-160	160	Hash	[20]
ripemd-256	-	Hash	[21]
ripemd-320	320	Hash	[22]
sha-1	160	Merkle-Damgard construction [13]	[23]

(*continued*)

Table 9-2. (*continued*)

Name	Length of the Tag (Bits)	Type	References
sha-224	224	Merkle-Damgard construction [13]	[25]
sha-256	256	Merkle-Damgard construction [13]	[24]
sha-384	384		[24]
sha-512	512		[24]
sha-3 (keccak)	Arbitrary	Sponge functions [14]	[26], [27]
whirlpool	512	Hash	[28], [29]

Hash functions have good applicability in digital signatures. Usually, when working with digital signatures, the hash value of a long message is computed; only the hash value is signed, and the sender sends it to the receiver. On the other hand, the receiver computes the hash value of the received message. If it corresponds to the one sent by the sender, then the message is righteous. This approach saves time and space complexity because, otherwise, the original message should be split into blocks of a predefined length, and then each block needs to be signed.

The BLAKE family [15] of hash functions uses the ChaCha stream cipher [30] as its mathematical basis. The modifications besides ChaCha consist of adding a permuted version of the input block XORed with constant numbers of the current round before each ChaCha round. In BLAKE, there are used hash values of 8 words, which are combined with 16 message words, and then the ChaCha value is truncated, obtaining the next hash value.

The GOST hash function works as follows: the input message is split into blocks of 256 bits, and then a number of 0s are padded to the message such that it reaches the length of 256 bits. The bits that are remaining are replaced by the sum of all previous hashed blocks represented on 256 bits and an integer value represented on 256 bits that is the length of the original message. However, [31] describes a collision attack that breaks it into 2^{105} time.

The father of the message digest (MD) hash functions is Ronald Rivest, who proposed MD2 in 1989. MD2-MD5 hash functions work on 128 bits. Each represents an improvement of the previous one, and their mathematical construction is based on the Merkle-Damgard function. Although on time, it was proven that they are not resistant

to collision attacks. For example, MD5 is still used as a checksum for data integrity, although it is vulnerable to length extension attacks [32]. However, MD hash functions represent the basis for other hash functions, such as RIPEMD. MD6 uses another approach, namely, the Merkle tree. Their authors show that MD6 resists different types of differential attacks [33].

The RACE Integrity Primitives Evaluation Message Digest (RIPEMD) family of functions are created based on MD4 and MD5, and their performance is close to the one of SHA-1.

The Secure Hashing Algorithms (SHA) family of hash functions is issued by the National Institute of Standards and Technology (NIST). SHA-1 was designed by the National Security Agency (NSA) after it was proved that the original SHA was insecure. The basis of SHA-1 lies in the MD5 hash function. However, it was proved that SHA-1 is also insecure. SHA-224, SHA-256, SHA-384, and SHA-512 are included in the next iteration of the SHA functions (known as SHA-2), which were also designed by the NSA. SHA-3 hash functions represent the results of authors from outside of the NSA. These are a subset of the Keccak family primitives, whose mathematical basis lies in sponge functions. The reference [34] presents a well-organized table regarding the security of the SHA hash functions. Here it is shown that SHA-512 and SHA-3 are secure against collision attacks and length extension attacks.

The whirlpool hash function outputs a digest of 512 bits, and its mathematical basis lies on the square block cipher, which was also used to design the AES encryption system.

The most popular hash functions are those from the SHA-2 family. Although the MD family and others like it are not secure anymore, they can be widely used as checksum functions.

Although the hash functions are widely used with applications such as securing passwords in databases, checking the integrity of different types of files, signing in digital signatures, etc., they have a more recent important application, namely, the blockchain. The entire infrastructure of blockchain technology lies in hash functions. Blockchain, which is also called a *distributed ledger*, can be seen as a decentralized database; therefore, there is no central authority that manages the database. All nodes within the network are communicating on a peer-to-peer basis, which means that one node communicates with all of the rest directly. One of the elements that a block of data contains is the hash value of the previous block. With different types of consensus algorithms that establish which node can add a block in the blockchain and this

approach of the structure, the data within a blockchain structure is extremely difficult to forge. Figure 9-2 shows the general structure of a chain. Observe how the arrows indicate that the field Prev contains the hash value of the previous data block. For example, Bitcoin uses SHA-256 as its hash function.

Figure 9-2. *The structure of a chain*

In Java, using a hash function is easy, because they are already implemented in Java libraries (see more details in Chapter 2). In Listing 9-1 we provide a simple example of applying SHA-512 over input text. Figure 9-3 shows the result. Note that even a slight modification of the input text yields a very different hash value. In Figure 9-3 the word *hash* is written with a lowercase *h*, while in Figure 9-4, the word *Hash* is written with an uppercase *H*.

Listing 9-1. Example of Applying SHA-512 on Input Text

```
import java.security.MessageDigest;
import java.util.HexFormat;
import java.util.Scanner;

public class HashFunctionExample {
    public static void main(String[] args) throws Exception {
        System.out.println("\nType the text:");

        try (Scanner scn = new Scanner(System.in)) {
           var input = scn.nextLine();
           System.out.println("\nThe input text: " + input);

           var outputSha = MessageDigest.getInstance("SHA-512");
           outputSha.update(input.getBytes());

           byte[] digest = outputSha.digest();
```

```
17
18              var hexDigest = HexFormat.of().formatHex(digest);
19              System.out.println("The hex representation: " + hexDigest);
20          }
21      }
22  }
```

The code from Listing 9-1 is optimized as follows:

1. **Use `java.util.HexFormat` for Byte-to-Hex Conversion (lines 2 and 18):** HexFormat offers a cleaner way to convert byte arrays to hex strings. This reduces the need for manual conversion and improves readability.
2. **Try-with-resources on `Scanner` to Ensure Proper Closure (line 9):** This language feature automatically manages resources such as files and scanners, ensuring they are correctly closed after use, reducing the risk of resource leaks.
3. **`var` for Local Variables (lines 10, 13, 18):** Use the `var` keyword for local variables where the compiler can infer the type, making the code cleaner and less verbose.

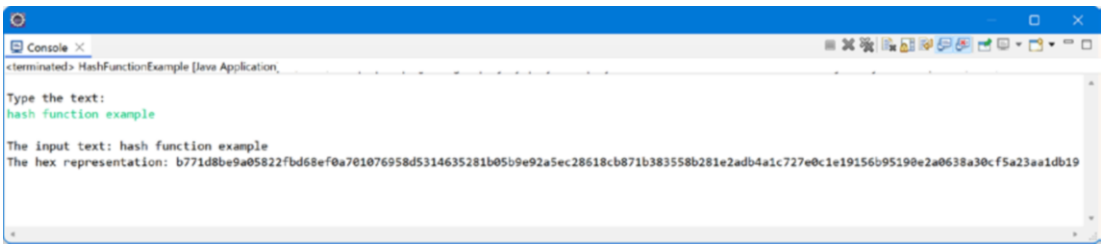


Figure 9-3. *The result of Listing 9-1 for the input "hash function example"*

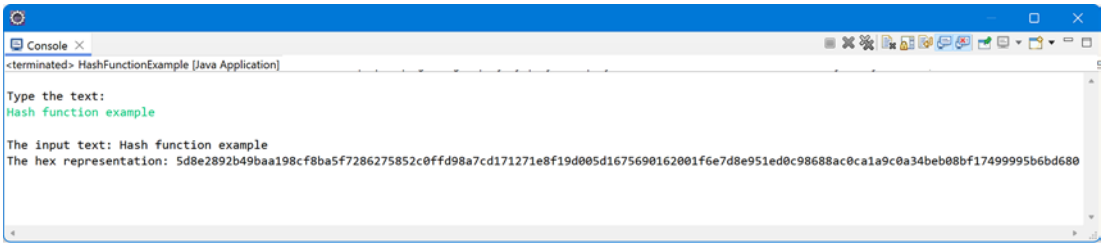


Figure 9-4. *The result of Listing 9-1 for the input "Hash function example"*

The three properties of hash functions (pre-image resistance, second pre-image resistance, collision resistance) give the security level of hash functions. These ensure that if an attacker modifies the input value (even a slight change), then the hash value of the input is also modified. Thus, this shows if the original value has integrity untouched. Different types of collisions can occur [36].

- **Near-collision.** For two input values, only some parts of the corresponding hash values are common.
- **Start-collision.** There can be various types of start conditions that are changed, for example, an initial value.

Besides collision resistance, hash functions must also meet two other security requirements.

- **Indistinguishability.** An attacker cannot distinguish a cryptographic primitive (for example, a block cipher) from an ideal one (for example, a permutation).
- **Indifferentiability.** This type of security is more suitable for hash functions than indistinguishability, due to hash function constructions and purpose. Indifferentiability targets the interaction between cryptographic primitives.

In [35] and [36], the authors classify attacks on hash functions into two categories.

- **Generic attacks.** These attacks target different properties of a hash function, except for collision resistance. Once a vulnerability is found, then the attacker can exploit it. These types of attacks do not depend on the details of the hash function's construction. Examples of such attacks are exhaustive search (an input value is chosen, and then the attacker searches through all possible hash values to find a match), (memoryless) birthday attacks, time-memory trade-off attacks (here, given a value, an attacker can compute the preimage), and meet-in-the-middle (this attack can be launched on an easily invertible hash function).
- **Cryptanalytic attacks.** These types of attacks target the compression function that is the foundation of a hash function. Such attacks have been launched for MD5 and SHA-1. Examples of such attacks are fixed-point attacks (which means that for a given message, if an attacker finds a component of the message, then the message without

that component has the same hash value as the original one), length-extension attacks (this attack involves computing the hash function of an extended version of an original message), multicollision attacks (based on the fact that more input values have the same hash value), and herding attacks (the attacker can compute the preimage of an unknown yet hash value).

Another common use case for hash functions is to create a digital fingerprint of files, ensuring data integrity. As we have seen, hashing can detect even minor changes in large data sets or files. This is particularly useful in scenarios like verifying software downloads, checking if files have been altered or corrupted, and in version control systems to track changes. Another example of a scenario is when handling highly confidential documents, for example, contracts, legal briefs, client information, or proprietary strategies. Given the sensitive nature of these examples of documents, any unauthorized alteration or corruption could lead to severe legal, financial, and reputational damage. Therefore, the integrity of these digital files should be checked. In Listing 9-2, we present a simple example of checking file integrity based on the SHA-256 hash function, while Figure 9-5 shows the result. To ensure file integrity, it is recommended to store the hash value of the file at first interaction, and check its integrity by computing the hash value before each subsequent use.

Listing 9-2. Computing the Hash Value of a File Using SHA-256

```
1   import java.nio.file.Files;
2   import java.nio.file.Paths;
3   import java.security.MessageDigest;
4   import java.util.HexFormat;
5
6   public class FileIntegrityChecker {
7
8       public static String calculateFileHash(String filePath) throws
        Exception {
9
10          var fileContent = Files.readAllBytes(Paths.get(filePath));
11          var digest = MessageDigest.getInstance("SHA-256");
12          var hashBytes = digest.digest(fileContent);
13
```

```
14          return HexFormat.of().formatHex(hashBytes);
15      }
16
17      public static void main(String[] args) throws Exception {
18          var filePath = "E:/Files/TestFile.txt";
19          var fileHash = calculateFileHash(filePath);
20
21          System.out.println("SHA-256 Hash: " + fileHash);
22      }
23  }
```

The code works similarly to the one from Listing 9-1; the difference here is the input, which is a file instead of a text. To work with files, `java.nio.file.Files` and `java.nio.file.Paths` classes are used. These are part of the `NIO.2` update, offering improved file I/O operations over the older `java.io.File` class.

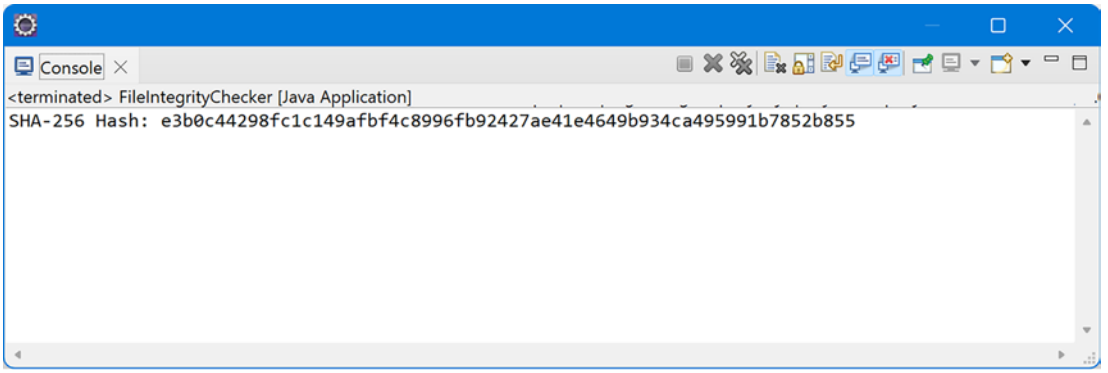


Figure 9-5. *The result of Listing 2 for the input file located at the path "E:/Files/TestFile.txt"*

Conclusion

In this chapter, hash functions were presented. Hash functions are widely used cryptographic primitives, and they have a large number of applications in the field of information security.

A newer use case for hash functions is blockchain technology, which bases its core on hash functions.

References

[1] Atanasiu, A. (2009). Securitatea Informatiei vol 2 (Protocoale de securitate). INFODATA Cluj Publishing House.

[2] Mihailescu, M. I., & Nita, S. L. (2021). Pro Cryptography and Cryptanalysis: Creating Advanced Algorithms with C# and. NET. Apress.

[3] Benaloh, J. (Ed.). (2014). Topics in Cryptology--CT-RSA 2014: The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014, Proceedings (Vol. 8366). Springer.

[4] BLAKE2 Official Implementation. Available: https://github.com/BLAKE2/BLAKE2.

[5] Blake3. Available: https://github.com/BLAKE3-team/BLAKE3/.

[6] H. Krawczyk, M. Bellare, R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, 1997.

[7] API KMAC. Available: www.cryptosys.net/manapi/api_kmac.html.

[8] Kelsey, J., Chang, S. J., & Perlner, R. (2016). SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. NIST special publication, 800, 185.

[9] Rivest, R. L., Agre, B., Bailey, D. V., Crutchfield, C., Dodis, Y., Fleming, K. E., ... & Yin, Y. L. (2008). The MD6 hash function–a proposal to NIST for SHA-3. Submission to NIST, 2(3), 1-234. Available online: http://groups.csail.mit.edu/cis/md6/submitted-2008-10-27/Supporting_Documentation/md6_report.pdf.

[10] PMAC. Available online: https://web.cs.ucdavis.edu/~rogaway/ocb/pmac.htm.

[11] UMAC. Available online: http://fastcrypto.org/umac/.

[12] Biham, E., & Dunkelman, O. (2007). A Framework for Iterative Hash Functions—HAIFA (No. CS Technion report CS-2007-15). Computer Science Department, Technion.

[13] Damgård, I. B. (1989, August). A design principle for hash functions. In Conference on the Theory and Application of Cryptology (pp. 416–427). Springer, New York, NY.

[14] Gilbert, H., & Handschuh, H. (2003, August). Security analysis of SHA-256 and sisters. In International workshop on selected areas in cryptography (pp. 175–193). Springer, Berlin, Heidelberg.

[15] BLAKE-256. Available online: https://docs.decred.org/research/blake-256-hash-function/.

[16] GOST. Available: https://tools.ietf.org/html/rfc5830.

[17] Rivest, R. L. (1990, August). The MD4 message digest algorithm. In Conference on the Theory and Application of Cryptography (pp. 303-311). Springer, Berlin, Heidelberg.

[18] Rivest, R., & Dusse, S. (1992). The MD5 message-digest algorithm.

[19] RIPEMD-128. Available online: https://homes.esat.kuleuven.be/~bosselae/ripemd/rmd128.txt.

[20] RIPEMD-160. Available online: https://homes.esat.kuleuven.be/~bosselae/ripemd160.html.

[21] Yeh, Y. S., & Chou, J. S. (2001). Keyed/Unkeyed RIPEMD-128, 192, 256. Journal of Information and Optimization Sciences, 22(3), 563–578.

[22] Sasaki, Y., & Aoki, K. (2009, July). Meet-in-the-middle preimage attacks on double-branch hash functions: Application to RIPEMD and others. In Australasian Conference on Information Security and Privacy (pp. 214–231). Springer, Berlin, Heidelberg.

[23] Wang, X., Yin, Y. L., & Yu, H. (2005, August). Finding collisions in the full SHA-1. In Annual international cryptology conference (pp. 17–36). Springer, Berlin, Heidelberg.

[24] Descriptions of SHA-256, SHA-384, and SHA-512. Available online: www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf.

[25] A 224-bit One-way Hash Function: SHA 224. Available online: https://datatracker.ietf.org/doc/html/rfc3874.

[26] Paul Hernandez, "NIST Releases SHA-3 Cryptographic Hash Standard," 2015.

[27] Morris J. Dworkin, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions". Federal Inf. Process. STDS. (NIST FIPS) – 202. 2015.

[28] Paulo S. L. M. Barreto "The WHIRLPOOL Hash Function". 2008. Archived from the original on 2017-11-29. Retrieved 2018-08-09.

[29] Paulo S. L. M. Barreto and Vincent Rijmen, "The WHIRLPOOL Hashing Function." 2003. Archived from the original (ZIP) on 2017-10-26. Retrieved 2018-08-09.

[30] Bernstein, D. J. (2008, January). ChaCha, a variant of Salsa20. In Workshop record of SASC (Vol. 8, pp. 3–5).

[31] Mendel, F., Pramstaller, N., Rechberger, C., Kontak, M., & Szmidt, J. (2008, August). Cryptanalysis of the GOST hash function. In Annual International Cryptology Conference (pp. 162-178). Springer, Berlin, Heidelberg.

[32] Stevens, M. (2006). Fast Collision Attack on MD5. IACR Cryptol. ePrint Arch., 2006, 104.

[33] Rivest, R. L., Agre, B., Bailey, D. V., Crutchfield, C., Dodis, Y., Fleming, K. E., ... & Yin, Y. L. (2008). The MD6 hash function–a proposal to NIST for SHA-3. Submission to NIST, 2(3), 1-234.

[34] Secure Hash Algorithms, Available online: https://en.wikipedia.org/wiki/Secure_Hash_Algorithms.

[35] Gauravaram, P., & Kelsey, J. (2007). Cryptanalysis of a class of cryptographic hash functions. Cryptology EPrint Archive.

[36] Toz, D. (2013). Cryptanalysis of Hash Functions (Cryptanalyse van hashfuncties).

CHAPTER 10

Symmetric Encryption Algorithms

In this chapter, we will discuss two types of symmetric encryption algorithms, Data Encryption Standard (DES) and Advanced Encryption Standard (AES). The goal of this chapter is to provide a starting point for implementing such algorithms from scratch without using third-party libraries, by using the theoretical concepts and translating them into a practical application using Java.

The transition from the DES to the AES is a significant development in cryptography, demonstrating the connection between evolving cryptographic technologies and the need for enhanced security measures to address new threats.

The DES encryption technique officially started being used in 1977, and its development was inspired by previous efforts, such as the Lucifer cipher created by Horst Feistel at IBM. The U.S. National Bureau of Standards, now known as the National Institute of Standards and Technology (NIST), searched for a strong encryption standard to protect nonclassified government communications. This led to the selection of DES after a public contest for algorithms and adjustments recommended by the National Security Agency (NSA) to enhance its resistance to differential cryptanalysis, which was revealed only years later.

The Data Encryption Standard (DES) works as a block cipher using a 56-bit key to encrypt 64-bit blocks of data. At first, this was considered secure. However, with the advancement of computational power, the 56-bit key length, chosen for a balance between security and computational efficiency, became vulnerable to brute-force attacks. By the late 1990s, because of significant developments in computing technology, DES's weaknesses became feasible rather than just theoretical. In 1997, the DESCHALL Project effectively showed a brute-force attack against DES, showing that DES could be cracked within a reasonable time frame, making the standard ineffective for many high-security applications.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_10

Triple DES (3DES) was introduced to address increasing concerns about the security of DES. Triple Data Encryption Standard (3DES) involves using the Data Encryption Standard (DES) encryption algorithm three times for each data block, which enhances security by essentially extending the key length. Although improved, 3DES maintained DES's block size and structure, allowing for a simple but slower upgrade from the original DES.

Another call for proposals was launched to find a more efficient and secure alternative to DES and 3DES. In 1997, NIST initiated the AES development project and called upon international cryptographers to submit their proposals. In 2001, after rigorous testing and evaluation involving the global cryptographic community, NIST selected the Rijndael cipher, developed by Belgian cryptographers Vincent Rijmen and Joan Daemen, as the new AES. Rijndael was chosen for its combination of security, performance, efficiency, practicability, and flexibility. AES supports key sizes of 128, 192, and 256 bits, compared with DES's fixed 56-bit key, making AES secure against all forms of brute-force attacks designed in the age of classical computing.

AES has been widely incorporated in all areas of cybersecurity, being the best option for symmetric encryption. Its application in government encryption protocols and private-sector data security highlights its adaptability and strength. AES is important in protocols like SSL/TLS for secure Internet communications, for wireless security in WPA2, and in different applications that need to encrypt sensitive data.

Data Encryption Standard

This section will describe DES as it was published by NIST in publication FIPS PUB 46-3 [5]. DES is a block cipher that is included in the symmetric encryption systems category, as it uses just one secret key for the encryption and for the decryption. At this point, DES should not be used in real applications because it has been cracked, and the literature is rich with different cryptanalysis attacks [4].

DES is an algorithm that has its origin in the LUCIFER algorithm, also known as the Feistel block cipher algorithm. It was developed by Horst Feistel at IBM, a brilliant researcher. Its security was based on the secret key represented in blocks of 128 bits and another block also represented on 128 bits. Another important aspect that is related to its design is the fact that the algorithm uses 16 rounds within the Feistel structure. Each round has its own key.

In 1976, it was recognized and adopted as a Federal Encryption Standard. In 1995, an advanced version of the DES algorithm, known as Triple DES (for short, 3DES or TDES), was proposed. Starting with 2002, AES (which will be discussed later) was released, and it was used to replace the DES encryption algorithm as a recognized and adopted standard. Its official name is the Triple Data Encryption Algorithm (for short, TDEA or 3DEA).

TDEA represents another symmetric-key block algorithm that is based on the DES algorithm. For each data block, TDEA will apply DES three times. The size of the block is set to 64 bits, and the sizes of the keys are 56 bits, 112 bits, and 168 bits. Each key is represented as 1, 2, and 3, as shown in Figure 10-1. The rounds are equivalent to the ones from DES, with 48 rounds in total, which means 16 rounds for each key (16 × 3 = 48 rounds).

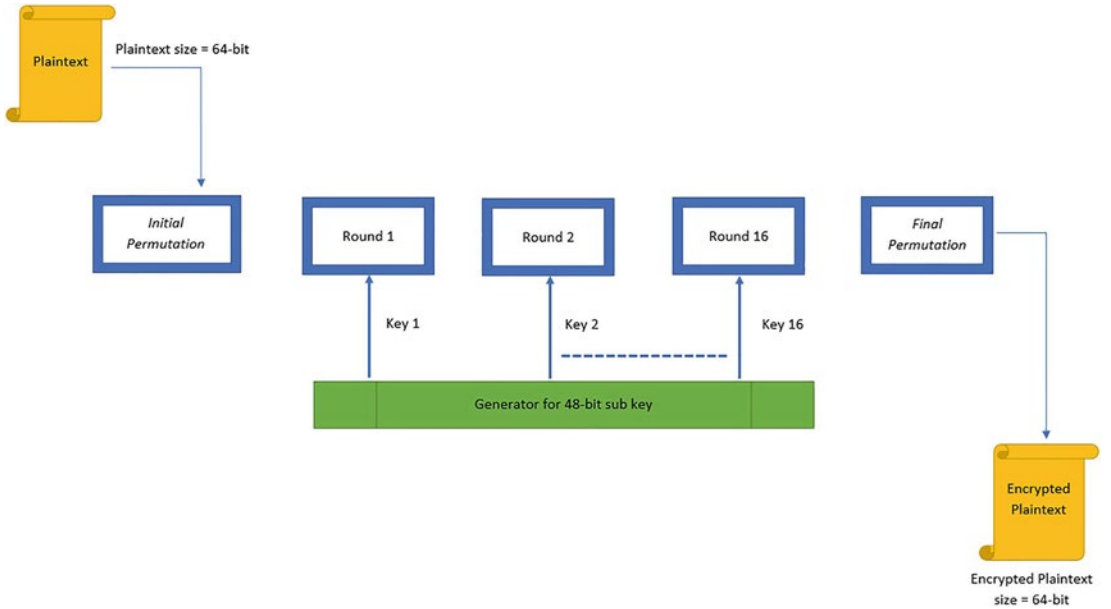


Figure 10-1. *Encryption process for 64-bit plaintext*

According to Figure 10-1, DES will encrypt the data using the first key (Key 1), and the decryption will be done using the second key (Key 2). Further, the encryption is performed again by using the third key (Key 3). There is another version of the algorithm in which only the first and third keys are used, and the keys are the same. Some applications still use this algorithm, but it is treated as a legacy algorithm.

Starting with Figure 10-2, you can see how the encryption and decryption processes are performed within TDEA.

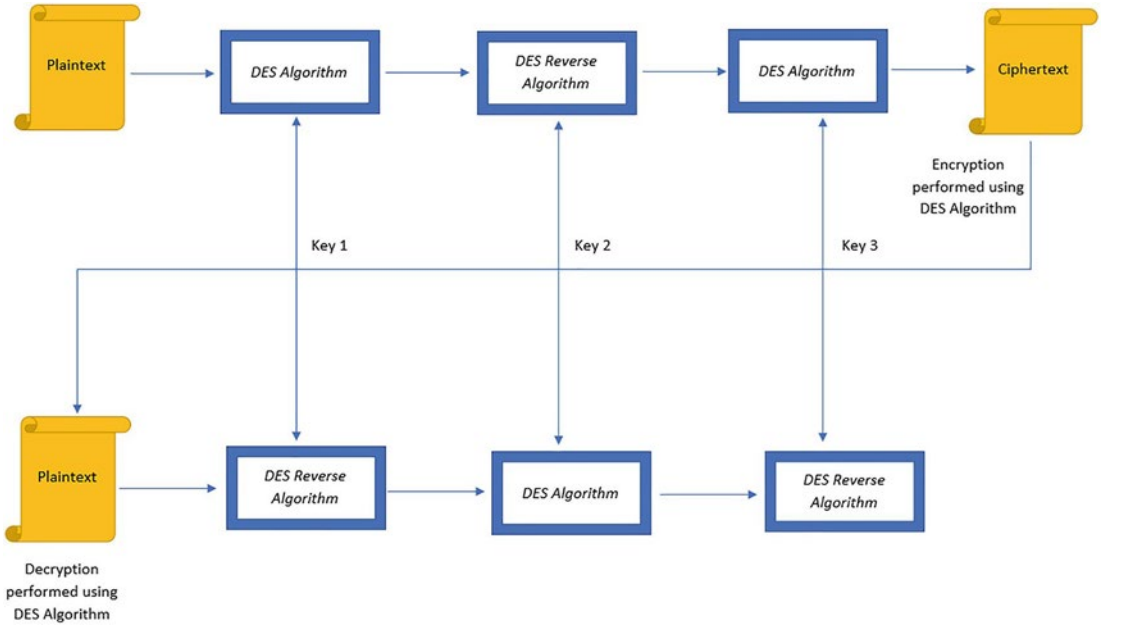


Figure 10-2. *Encryption and decryption process using TDEA*

At this point, we will not focus on the mathematical background, and we will proceed further with the implementation process. We strongly recommend following a theoretical course in cryptography and information security to go deep and understand what is happening during the implementation process. To achieve this goal, you'll find good references at the end of the chapter that will provide the best knowledge for a cryptography mathematical apparatus.

The Generation of Keys

The DES algorithm will compute 16 rounds of the encryption process for each of the rounds a genuine and unique key is generated. To continue with the next steps, it is important to understand how the bits representation of the plaintext are labeled, from 1 to 64. The most significant bit is represented by 1, and the least significant bit is represented by 64.

Let's take a look at the process of generating the keys. For this process, you need the PC-1 and PC-2 permutations; see Table 10-1.

Table 10-1. *Permutation PC-1 [5]*

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

To transform the permutation PC-1 or PC-2 from Table 10-1 or Table 10-2 in a Java expression, it is sufficient to declare a variable, for example, `int pc_1[56]` and `int pc2[48]`, respectively, and then initialize them properly as an array with a size set to 56 and 48, respectively.

Table 10-2. *Permutation PC-2 [5]*

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Encryption and Decryption Process

To perform the encryption process, the following steps should be followed to preserve the logic during the implementation process:

- Input
  - Data ← plaintext
  - Size ←64 – bit size

- The plaintext is converted to an initial permutation (IP) function.
- The IP will divide the plaintext into two equal-sized permuted blocks. In some documentation, these permuted blocks are called the Left Plain Text (LPT) and the Right Plain Text (RPT).
- Using the two blocks generated, LPT and RPT, the 16 rounds related to the encryption process are performed. The steps behind the 16 rounds are as follows:
  - Step 1: Key transformation
  - Step 2: Expansion permutation
  - Step 3: S-box permutation
  - Step 4: P-box permutation
  - Step 5: XOR
  - Step 6: Swap
- Once the encryption process is done, the two blocks (LPT and RPT) are rejoined.
- In the end, we get the encrypted version of the plaintext that will have a size of 64 bits.

The decryption process is based on using the same algorithm but in reverse order.

Another important aspect that is taken into consideration when implementing the DES algorithm is its modes of operation.

Operation Modes for DES

DES contains five modes of operation from which a developer can choose during the implementation process.

- **Cipher Block Chaining (CBC).** In this mode, each block is strictly connected with the previous node. It is using an initialization vector (IV).
- **Electronic Codebook (ECB).** For each block with a 64-bit size, the encryption and decryption are applied independently.

- **Cipher Feedback (CFB).** Once the ciphertext is obtained, it will be considered as the input for the algorithm. A pseudorandom output will be obtained from this model. The resulting output is computed using an XOR operation of the plaintext. Based on the XOR operation, the encryption is computed, and its result is passed further to the next operation.
- **Counter (CTR).** For each of the blocks from the plaintext, the XOR operation is performed with the help of an encrypted counter. The next step is quite tricky as it needs to be incremented sequentially for each of the blocks.
- **Output Feedback (OFB).** There are some similarities with the CFB mode. The single difference is represented by the input of the encryption algorithm, which is the output that precedes DES.

At this moment, we can say that we have covered all the elements and main concepts to perform the implementation process.

In Listing 10-1 we will provide a basic implementation for the DES algorithm in which we will show all the steps.

Listing 10-1. DES Algorithm (Encryption and Decryption)

```
1   package desalgorithm;
2
3   import java.io.*;
4   import java.security.spec.AlgorithmParameterSpec;
5   import javax.crypto.*;
6   import javax.crypto.spec.IvParameterSpec;
7
8   public class DES {
9       //instance for encryption
10      private static Cipher encOp;
11
12      //instance for decryption
13      private static Cipher decOp;
14
```

```
15      //path for the file that will be encrypted
16      private static final String textFileToEncrypt = "E:/Files/
        apressFile.txt";
17
18      //path for the encrypted file
19      private static final String encFile = "E:/Files/apress_enc.txt";
20
21      //path for decryptied file
22      private static final String decFile = "E:/Files/apress_dec.txt";
23
24      //vector for initialization
25      private static final byte[] iv = { 25, 38, 15, 43, 59, 92,
        66, 73 };
26
27      public static void main(String[] args) {
28          try {
29              //setting up the key
30              SecretKey secret_key = KeyGenerator.getInstance("DES").
                generateKey();
31              AlgorithmParameterSpec parameters_specs = new
                IvParameterSpec(iv);
32
33              //specify the encryption mode
34              encOp = Cipher.getInstance("DES/CBC/PKCS5Padding");
35              encOp.init(Cipher.ENCRYPT_MODE, secret_key,
                parameters_specs);
36
37              //specify the decryption mode
38              decOp = Cipher.getInstance("DES/CBC/PKCS5Padding");
39              decOp.init(Cipher.DECRYPT_MODE, secret_key,
                parameters_specs);
40
41              //encrypt the file
42              processFile(encOp, new File(textFileToEncrypt), new
                File(encFile));
```

```
43
44              //decrypt the file
45              processFile(decOp, new File(encFile), new File(decFile));
46
47              System.out.println("The files for encryption and decryption
                results have created successfully.");
48          } catch (Exception e) {
49              e.printStackTrace();
50          }
51      }
52
53      private static void processFile(Cipher cipher, File inputFile, File
        outputFile) throws IOException {
54          try (InputStream in = new FileInputStream(inputFile);
55               OutputStream out = new FileOutputStream(outputFile);
56               CipherOutputStream cipherOut = new CipherOutputStream(out,
                 cipher)) {
57              byte[] buffer = new byte[512];
58              int numRead;
59              while ((numRead = in.read(buffer)) >= 0) {
60                  cipherOut.write(buffer, 0, numRead);
61              }
62          }
63      }
64  }
```

Before running the implementation from Listing 10-1, we will need to have a file created at a specified location (in this example, `E:/Files/apressFile.txt`), such as the one specified in line 16 (see Figure 10-3 for its content). If the file does not exist, an exception is thrown with an error message (line 55). The other two files, for encryption and decryption, are created automatically.

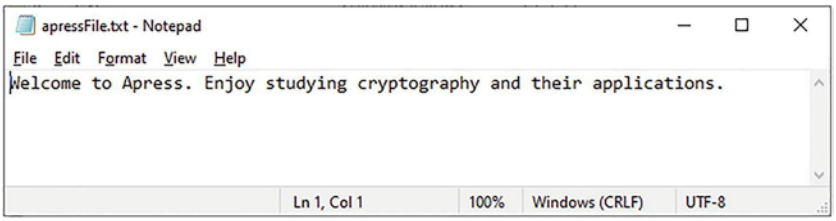


Figure 10-3. *The file and its content that will be encrypted*

Based on the file specified in line 16, when the encryption (see line 42) and decryption operations (see line 45) are performed, two files will be generated: `apress_enc.txt` for encryption output (see line 19 and Figure 10-4) and `apress_dec.txt` for decryption output (see line 22 and Figure 10-5).

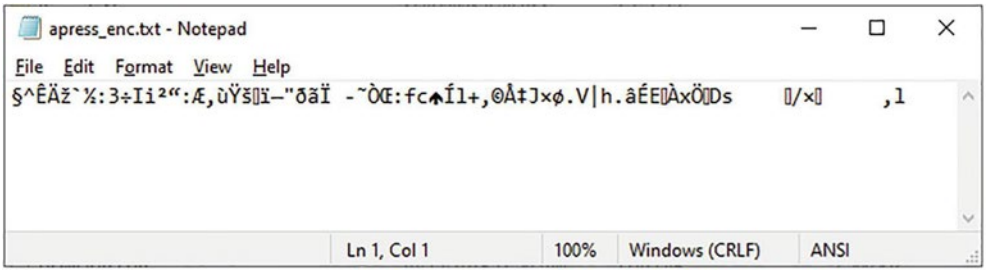


Figure 10-4. *File apress_enc.txt*

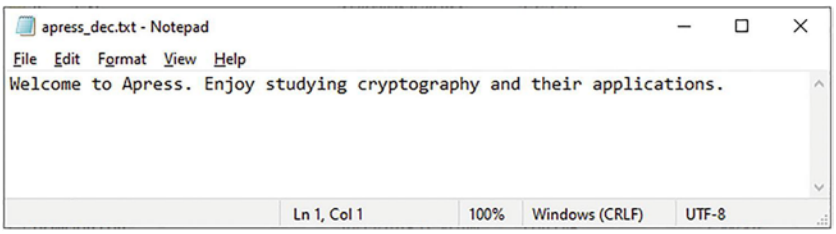


Figure 10-5. *File apress_dec.txt*

The following are the key features of the code from Listing 10-1:

- **Try-with-Resources.** Used for automatically closing both input and output streams (`InputStream, OutputStream`, and `CipherOutputStream`). This not only makes the code cleaner but also prevents resource leaks.

- **Refactoring File Processing.** Combined encryption and decryption processes into a single method called `processFile`, which takes a `Cipher` instance, input file, and output file as arguments. This method reads from the input file, processes the data through the cipher, and writes the output to the file, encapsulating the functionality that was previously duplicated in `ComputeEncOperation` and `ComputeDecOperation`.
- **Improved Error Handling.** By centralizing file processing, the error handling is also centralized, reducing code duplication and making the program more robust.
- **Eliminating Unnecessary File Existence Check.** Instead of explicitly checking if the file exists, allowing the `FileInputStream` to throw `FileNotFoundException` simplifies the code and is more in line with common Java practices.

Advanced Encryption Standard

The Advanced Encryption Standard represents a symmetric block cipher that provides encryption and decryption operations, published by NIST under the publication FIPS 197 [6]. This section will follow the technical description of AES as it is in [6]. The size of the key accepted is 128, 192, or 256 bits. The same key used for encryption has to be used during the decryption process as well.

An important aspect of AES during the encryption process is that the data is converted into a block, and then the encryption is made using padding operations with blocks of length 128, 192, or 256 bits.

Several advantages can be summarized as follows:

- The encrypted message/file can be decrypted without the key that has been used in the encryption stage. A valid secret key has to be used.
- AES is used in different areas of applications, such as data storage, financial transactions, wireless device communication, and much more.
- It is a secure algorithm, but some cryptanalysis attacks have been launched with success [3].

Also, there are some disadvantages, such as the following:

- The mechanism (mathematical apparatus) behind AES is quite simple. For further reading, I strongly recommend pursuing a cryptography course such as [1, 2].
- AES can represent a serious challenge during the implementation of the software.

Encryption and Decryption Process

During the encryption process, the plaintext will be encrypted using a secret key that is known only by the sender and by the receiver of the message.

Within Java, the encryption and decryption are represented by the Java Cryptographic Extension (JCE) platform. JCE offers different packages that can be used by developers during their implementation processes, such as `java.security` and `javax.crypto` (see lines 4–8 from Listing 10-2).

The same process that is used for encryption is used as well for decryption. With the help of the `init()` method from the `Cipher` class, the algorithm will be initialized with the public key based on the chosen transformation.

Operation Modes for AES

There are five modes of operation from which a developer can choose during the implementation process.

- **Cipher Block Chaining (CBC).** In CBC mode the initialization vector is used to get a better process of the encryption process. To obtain the encryption output, the CBC uses an XOR operation between the initialization vector and plaintext.
- **Electronic Codebook (ECB).** This is one of the simplest modes of all modes available. The plaintext is divided into blocks, with each block having 128 bits. The same key and algorithm chosen in the beginning are used to encrypt the blocks.

- **Cipher Feedback (CFB).** The first operation performed by CFB is to perform the encryption of the initialization vector. Once we have the encryption of the initialization vector, an XOR operation with the plaintext is performed, and the encrypted message is obtained. Moving forward, the encrypted message (or block) is encrypted with the next plaintext message (or block).
- **Counter (CTR).** For each of the blocks from the plaintext, the XOR operation is performed with the help of an encrypted counter. The next step is quite tricky as it needs to be incremented sequentially for each of the blocks.
- **Output Feedback (OFB).** There are some similarities with the CFB mode. The single difference is represented by the input of the encryption algorithm, which is the output that precedes DES.
- **Galois and Counter Mode (GCM).** This mode represents an extended version of CTR mode. It has been proposed and introduced by NIST. In GCM mode we have the ciphertext and an authentication label once the encryption process is done.

Based on the elements that we have mentioned, we can say that we have all the elements and main concepts set to perform with the implementation process.

In Listing 10-2 we will provide a basic implementation for the AES algorithm in which we will emphasize all the steps. Figure 10-6 shows the result.

Listing 10-2. Implementation of the AES Algorithm

```
1   package aesalgorithm;
2
3   import java.util.*;
4   import javax.crypto.*;
5   import javax.crypto.spec.*;
6   import java.security.spec.*;
7   import java.nio.charset.*;
8
9   public class AES {
10      private static final byte[] INITIALIZATION_VECTOR = {54, 34, 7, 3,
        23, 78, 31, 68, 32, 40, 96, 43, 23, 54, 23, 76};
```

```
11      private static final String AES_SECRET_KEY = "cryptoApress";
12      private static final String AES_SALT = "apress";
13
14      public static String encrypt(String plainMessage) {
15          try {
16              Cipher cipher = initCipher(Cipher.ENCRYPT_MODE);
17              return Base64.getEncoder().encodeToString(cipher.
                doFinal(plainMessage.getBytes(StandardCharsets.UTF_8)));
18          } catch (Exception e) {
19              throw new RuntimeException("Encryption error", e);
20          }
21      }
22
23      public static String decrypt(String encryptedMessage) {
24          try {
25              Cipher cipher = initCipher(Cipher.DECRYPT_MODE);
26              return new String(cipher.doFinal(Base64.getDecoder().
                decode(encryptedMessage)), StandardCharsets.UTF_8);
27          } catch (Exception e) {
28              throw new RuntimeException("Decryption error", e);
29          }
30      }
31
32      private static Cipher initCipher(int mode) throws Exception {
33          IvParameterSpec ivParameterSpec = new
            IvParameterSpec(INITIALIZATION_VECTOR);
34
35          //the container for the secret key
36          SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBK
            DF2WithHmacSHA256");
37
38          //specification parameters (secret key, salt value, iterations,
            key length)
39          KeySpec keySpec = new PBEKeySpec(AES_SECRET_KEY.toCharArray(),
            AES_SALT.getBytes(), 65536, 256);
```

```
40
41          //generate the secret key based on the parameters set above
42          SecretKey tmp = keyFactory.generateSecret(keySpec);
43
44          //align the secret key with the AES algorithm
45          SecretKeySpec secretKeySpec = new SecretKeySpec(tmp.
            getEncoded(), "AES");
46
47          //set the algorithm (e.g., AES) and its mode together with its
            padding type
48          Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
49          cipher.init(mode, secretKeySpec, ivParameterSpec);
50
51          return cipher;
52      }
53
54      public static void main(String[] args) {
55          //set the message that we want to encrypt
56          String originalValue = "Welcome to Apress. Enjoy learning
            cryptography";
57
58          //perform the encryption
59          String encryptedValue = encrypt(originalValue);
60
61          //perform the decryption
62          String decryptedValue = decrypt(encryptedValue);
63
64          //show some messages
65          System.out.println("Plaintext used for encryption and
            decryption -> " + originalValue);
66          System.out.println("The encryption is -> " + encryptedValue);
67          System.out.println("The decryption is -> " + decryptedValue);
68      }
69  }
```

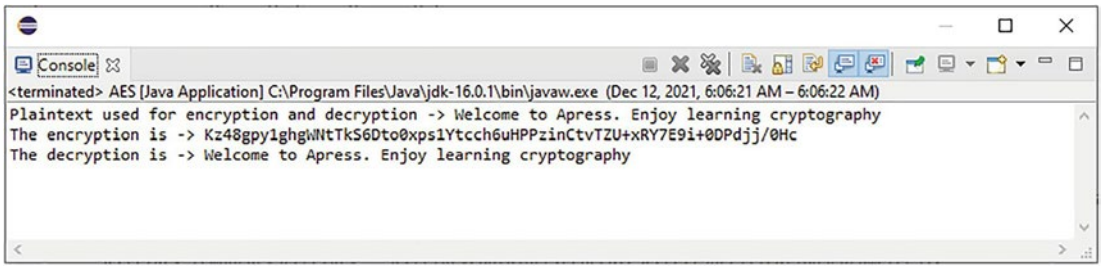

Console
<terminated> AES [Java Application] C:\Program Files\Java\jdk-16.0.1\bin\javaw.exe (Dec 12, 2021, 6:06:21 AM – 6:06:22 AM)
Plaintext used for encryption and decryption -> Welcome to Apress. Enjoy learning cryptography
The encryption is -> Kz48gpy1ghgWNtTkS6Dto0xps1Ytcch6uHPPzinCtvTZU+xRY7E9i+0DPdjj/0Hc
The decryption is -> Welcome to Apress. Enjoy learning cryptography

Figure 10-6. *The output of encryption/decryption with AES*

Before running the implementation from Listing 10-2, we need to set the initialization vector, which can be set manually (see line 10); the secret key (see line 13); and the salt value (see line 15). As a best practice, we strongly recommend that the initialization vector be generated randomly without hard-coding it. This method is used only for the example and educational purposes. The encryption function works as follows: the function first initializes a `Cipher` instance for encryption mode using `initCipher(Cipher.ENCRYPT_MODE)` (line 16). This setup involves configuring the AES cipher with an initialization vector, the secret key derived from the password and salt, and the specified encryption mode. The `initCipher` method creates a cipher instance configured for AES encryption in CBC mode with PKCS5 padding. It uses a key derived from the given password (`AES_SECRET_KEY`), a salt (`AES_SALT`), and an iteration count to make brute-force attacks more difficult. The key is derived using Password-Based Key Derivation Function 2 (PBKDF2) with HMAC-SHA-256, which securely generates a key from the password and salt. This derived key is then wrapped in a `SecretKeySpec` for use with AES. Then, the plaintext string is converted to bytes using UTF-8 encoding. These bytes are then encrypted using the configured `Cipher` object. The resulting encrypted bytes are then encoded into a Base64 string to ensure they are representable in text formats and printable characters, making it easier to store or transmit (line 17). The decryption process works similarly: the function initializes a `Cipher` instance, but this time in decryption mode using `initCipher(Cipher.DECRYPT_MODE)` (line 25). As in the encryption step, the `initCipher` method configures the cipher for AES decryption in CBC mode with PKCS5 padding, using the same derived key and initialization vector. The encrypted message, which is received as a Base64-encoded string, is first decoded from Base64 back into its original byte form. These bytes are then decrypted using the configured `Cipher` object. The decrypted bytes are converted back into a string using UTF-8 encoding to form the final plaintext output (line 26).

Conclusion

In this chapter, we discussed symmetric encryption algorithms such as Data Encryption Standard and Advanced Encryption Standard. You now should understand the main differences, advantages, and disadvantages of DES and AES, as well as a short summary of the modes used for each of the algorithms.

The examples provided were straightforward; we used a common and basic implementation and explained the mode in which they should be implemented. The implementations that we have provided are offered for educational purposes, and there is no guarantee that in practice they will offer a maximum benefit, especially in the case of the AES algorithm, which is quite sensitive about the process of implementation.

Lastly, Figure 10-7 summarizes the encryption process between DES and AES, as shown in Figure 10-7.

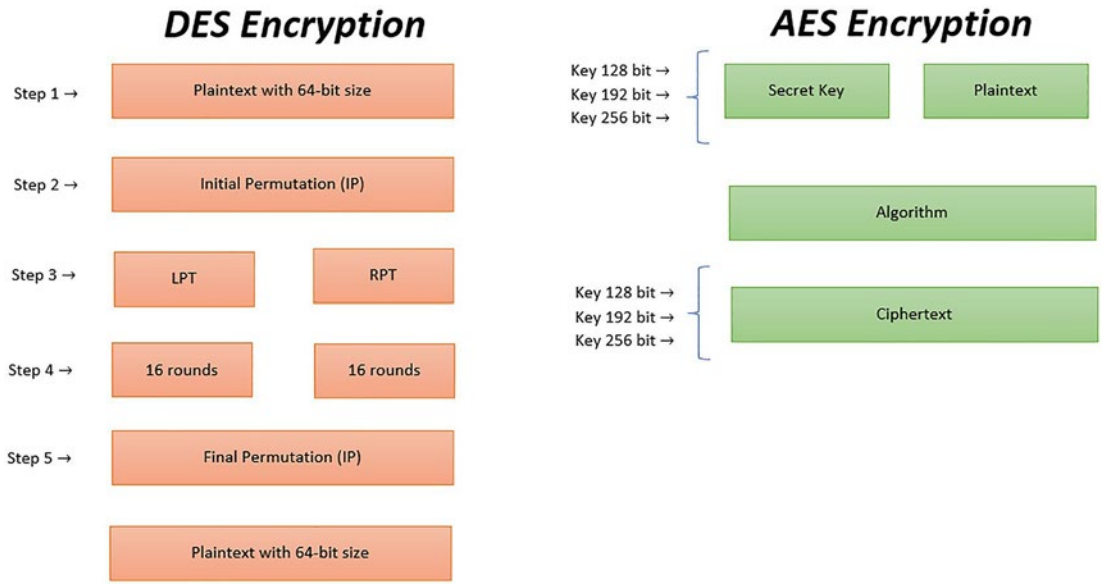


Figure 10-7. *Summary of the encryption process between DES and AES*

References

[1] Stinson, Douglas R., and Maura B. Paterson. Cryptography: Theory and Practice. Fourth edition, CRC Press, Taylor & Francis Group, 2019.

[2] Menezes, A. J., et al. Handbook of Applied Cryptography. CRC Press, 1997.

[3] Dobbertin H., Knudsen L., Robshaw M. (2005) The Cryptanalysis of the AES – A Brief Survey. In: Dobbertin H., Rijmen V., Sowa A. (eds) Advanced Encryption Standard – AES. AES 2004. Lecture Notes in Computer Science, vol 3373. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11506447_1.

[4] Biham, E., Shamir, A. Differential cryptanalysis of DES-like cryptosystems. *J. Cryptology* **4**, 3–72 (1991). https://doi.org/10.1007/BF00630563.

[5] FIPS PUB 46-3. Available online: https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf.

[6] FIPS PUB 197. Available online: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

CHAPTER 11

Asymmetric Encryption Schemes

Another name for asymmetric encryption is *public-key cryptography* (PKC). This name comes from the fact that the cryptosystems in this category are using a pair of keys in their algorithms. Specifically, a *public key* is used by the encryption algorithm, and a *private (secret) key* is used by the decryption algorithm. In classical cryptography, the purpose of each type of key is well established, but in modern cryptography, these types of keys can be used for additional purposes. For example, another type of secret key can be a master key, used to encrypt and decrypt the keys of a specific cryptosystem.

While in symmetric encryption schemes the cryptosystems use mathematical concepts such as substitutions and permutations (learn about the AES and DES algorithms in Chapter 10), asymmetric encryption has asymmetric functions as its foundations. The authors who introduced the concept of asymmetric encryption are Diffie and Hellman in their paper "New directions in cryptography" [1]. In this paper, the authors explained what asymmetric encryption is and what the characteristics of such a cryptosystem are. In addition, they showed how the secret keys can be securely distributed over an insecure channel between the sender and the receiver. Prior to this, all cryptographic systems were symmetric, meaning the same key was used for both encryption and decryption. This posed a substantial security risk and logistical problem—the—because sharing the key over insecure channels without compromising it *key distribution problem* was very challenging. Diffie and Hellman's breakthrough concept introduced a two-key system where each participant in a communication had a pair of keys: one public and one private. The public key could be shared openly, while the private key was kept secret. This innovation addressed the key distribution problem by allowing the public key to be freely distributed without compromising the security of the private key, which remained confidential and was used to decrypt the message.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_11

This new concept was welcomed by the scientific community, and many encryption systems that follow the rules from [1] were proposed in the meantime. The following are the requirements presented in [1] that need to be accomplished by the cryptosystems:

- The receiver (which is the owner of the secret key) should be able to use a computationally efficient algorithm to generate the pair of keys.
- A pair of keys is used. The public key is used to encrypt the plain messages, and the secret key is used to decrypt the encrypted messages.
- The sender should be able to use a computationally efficient algorithm to encrypt the plain text (message) into the encrypted text.
- The receiver should be able to use a computationally efficient algorithm to recover the plain text (message) from the encrypted text.
- The secret key should be independent of the public key; therefore, the public key cannot be used as input for the secret key generation.
- The plain text (message) cannot be computed if one knows the public key and the encoded message.

The introduction of asymmetric encryption had a revolutionary impact on data security for several reasons:

- **Secure key distribution.** Asymmetric encryption eliminated the need for sending sensitive encryption keys across the network. This dramatically reduced the risk of interception by malicious actors.
- **Digital signatures.** Asymmetric encryption facilitated the development of digital signatures, which verify the integrity of the data and the identity of the sender, thus providing nonrepudiation.
- **Enabling secure communication.** It allowed for secure communications between parties who had no prior contact or secure channel to exchange keys, broadening the possibilities for secure digital interactions across the globe.

Generally, an encryption system that uses a pair of keys as described works as follows between the sender (S) and the receiver (R):

> S and R generate a pair of public and private keys, making the public key available to everyone, while the private key is kept secret by each party.
>
> To send a message, S uses the public key of R and encrypts the message using the encryption algorithm.
>
> To recover the message, R uses its private key and the decryption algorithm to recover the message.

Some examples of well-known public-key encryption systems are RSA [2], ElGamal [3], and Merkle-Hellman [4], which will be explained individually in the remainder of this chapter. Asymmetric encryption is important because it is used in different applications in cryptography, such as digital certificates and signatures, protocols (encryption, multiparty computation, zero-knowledge), etc. Types of encryption schemes include asymmetric encryption, homomorphic encryption, searchable encryption, predicate encryption, functional encryption, etc., while branches of public-key cryptography are integer factorization cryptography, elliptic-curve cryptography, lattice-based cryptography, learning and ring-learning with errors, multivariate cryptography, code-based cryptography, etc.

Sometimes, symmetric cryptosystems are faster than asymmetric ones, although they may have the same security level. Usually, in these cases, the plain messages are encrypted using the symmetric encryption scheme, and the secret key is encrypted using the asymmetric encryption scheme and is sent between the sender and receiver over an insecure channel. For example, Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols utilize asymmetric encryption for secure Internet communications, providing a foundation for secure web browsing, email, and other data transfers.

RSA

A well-known and commonly used public-key cryptosystem is RSA [2], which was introduced in 1978 and named after its authors: Ron Rivest, Adi Shamir, and Len Adleman. The hardness assumption for RSA is integer factorization, which means that for two prime numbers that are large enough, their product is *hard* to be factorized by a computing machine. What specifically does hard mean? It means that the problem to be solved requires more than polynomial time.

In the remainder of this section, we will present the algorithms of the RSA cryptosystem [2] and then their implementation in Java.

The first algorithm is the *key generation* algorithm, and it works as follows:

1. Generate two distinct large prime numbers p, q.
2. Determine $n = pq$.
3. Determine $\phi(n) = (p - 1)(q - 1)$. The ϕ function is called Euler's totient function.
4. Generate the integer value e, with $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$.
5. Determine $d \equiv e^{-1}(mod\ \phi(n))$.

The public key of the cryptosystem is $k_p = (n, e)$, and the private (secret) key is $k_s = (p, q, \phi(n), d)$; thus, in the encryption algorithm, the values n and e will be used, and in the decryption algorithm the values p, q, $\phi(n)$, and d will be used.

The next algorithm is the *encryption algorithm*, which encrypts the plain message using the following formula. The plain message is taken as an integer value m, $0 \leq m < n$, and the encrypted value obtained is c:

$$m^e \equiv c \ \left(mod\ \ n\right)$$

The last algorithm is the *decryption algorithm*, which recovers the plain message m from the encrypted value c, using the following formula:

$$c^d \equiv \left(m^e\right)^d \equiv m \ \left(mod\ \ n\right)$$

Listing 11-1 presents the RSA algorithm in Java.

Listing 11-1. Implementation of the RSA Cryptosystem

```
1  package rsa;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.math.BigInteger;
7  import java.security.SecureRandom;
8
```

```
9   public class RSA {
10
11      private BigInteger prime_p, prime_q, val_n, phi_n, val_e, val_d;
12      private final BigInteger one = BigInteger.ONE;
13      private final int maximumLength = 1024;
14      private SecureRandom random;
15
16      public void KeyGeneration() {
17          random = new SecureRandom();
18          prime_p = BigInteger.probablePrime(maximumLength, random);
19          prime_q = BigInteger.probablePrime(maximumLength, random);
20          val_n = prime_p.multiply(prime_q);
21          phi_n = prime_p.subtract(one).multiply(prime_q.subtract(one));
22
23          val_e = BigInteger.probablePrime(maximumLength, random);
24          while (phi_n.gcd(val_e).compareTo(one) != 0 || val_e.
            compareTo(phi_n) >= 0) {
25              val_e = BigInteger.probablePrime(maximumLength, random);
26          }
27          val_d = val_e.modInverse(phi_n);
28      }
29
30      public byte[] Encryption(byte[] plainMessage, BigInteger e,
        BigInteger n) {
31          BigInteger messageBigInt = new BigInteger(plainMessage);
32          BigInteger encryptedMessage = messageBigInt.modPow(e, n);
33          return encryptedMessage.toByteArray();
34      }
35
36      public byte[] Decryption(byte[] encryptedMessage, BigInteger d,
        BigInteger n) {
37          BigInteger messageBigInt = new BigInteger(encryptedMessage);
38          BigInteger decryptedMessage = messageBigInt.modPow(d, n);
39          return decryptedMessage.toByteArray();
40      }
```

```
41
42      public static void main (String [] arguments) {
43          RSA rsa = new RSA();
44          rsa.KeyGeneration();
45
46          try (BufferedReader reader = new BufferedReader(new
            InputStreamReader(System.in))) {
47              System.out.print("Type the plain message: ");
48              String plainMessage = reader.readLine();
49
50              System.out.println("\nEncrypting the message... ");
51              byte[] encryptedMessage = rsa.Encryption(plainMessage.
                getBytes(), rsa.val_e, rsa.val_n);
52              System.out.println("Encrypted message [bytes]: " + byteTo
                String(encryptedMessage));
53              System.out.println("Encrypted message [text]: " + new
                String(encryptedMessage, java.nio.charset.StandardCharsets.
                UTF_8));
54
55              System.out.println("\nDecrypting the message... ");
56              byte[] decryptedMessage = rsa.Decryption(encryptedMessage,
                rsa.val_d, rsa.val_n);
57              System.out.println("Decrypted message [bytes]: " + byteTo
                String(decryptedMessage));
58              System.out.println("Decrypted message [text]: " + new
                String(decryptedMessage, java.nio.charset.StandardCharsets.
                UTF_8));
59          } catch (IOException e) {
60              e.printStackTrace();
61          }
62      }
63
64      private static String byteToString(byte[] byteArray) {
65          StringBuilder recoveredString = new StringBuilder();
66          for (byte byteVal : byteArray) {
```

```
67              recoveredString.append(Byte.toString(byteVal));
68          }
69          return recoveredString.toString();
70      }
71  }
```

The code from Listing 11-1 follows the steps of the RSA algorithms presented in this section. In lines 3–7, the program begins by importing the necessary Java packages: `BufferedReader` and `InputStreamReader` facilitate reading input from the console, `IOException` handles errors that may occur during input/output operations, `BigInteger` manages large integers that are fundamental for RSA operations since RSA works with numbers that are too large for standard data types (note that BigInteger is a Java class, and not a third-party one), `SecureRandom` is used to generate secure random numbers, crucial for cryptographic security, especially for generating prime numbers for RSA keys (see Chapter 8 for details about random numbers). Lines 11–14 contain the declaration of the main variables and constant used in the implementation: `prime_p` and `prime_q` are prime numbers, which are crucial for generating the RSA keys; `val_n` represents the modulus for both the public and private keys; `phi_n` is Euler's totient function result for `n`; and `val_e` and `val_d` are the public and private keys, respectively. The constants `one` and `maximumLength` are used within key generation; `one` represents the integer one, and `maximumLength` defines the bit length of the prime numbers, typically set to 1024 bits to ensure cryptographic strength. Next, the RSA operations are encapsulated in three methods (key generation, encryption, decryption). `KeyGeneration` (lines 16–27) generates the RSA keys. `prime_p` and `prime_q` are generated using `SecureRandom` to ensure they are secure. The modulus `n` and the totient `phi_n` are computed. The public key `e` is chosen to be coprime with `phi_n`, and the private key `d` is the modular inverse of `e` modulo `phi_n`. `Encryption` (lines 30–34) converts the plain message from bytes to a `BigInteger` and then encrypts it using the public key. The encrypted data is converted back to a byte array and returned. `Decryption` (lines 36–40) is similar to encryption but uses the private key to decrypt the message. This demonstrates the asymmetry in RSA, where encryption and decryption use different keys. The main execution is given in lines 42–62; the main method calls the RSA operations. It instantiates the RSA object and generates the keys, and it prompts the user to enter a plain message, which is then encrypted and decrypted. The results are printed to the console, demonstrating the entire process from plaintext to encrypted text and back to decrypted text, verifying the correctness of the RSA implementation. The `byteToString` method (lines 64–70)

converts byte arrays into a readable string format, primarily for display purposes. The size for the keys is 1024 (given by the value of `maximumLength`); in practice, the key size of RSA is 512, 1024, or even 2048. Figure 11-1 shows the output of Listing 11-1. Note that the values obtained in Figure 11-1 can be different every time because pseudorandom numbers are used. However, the plain message should be the same as the decrypted message every time.

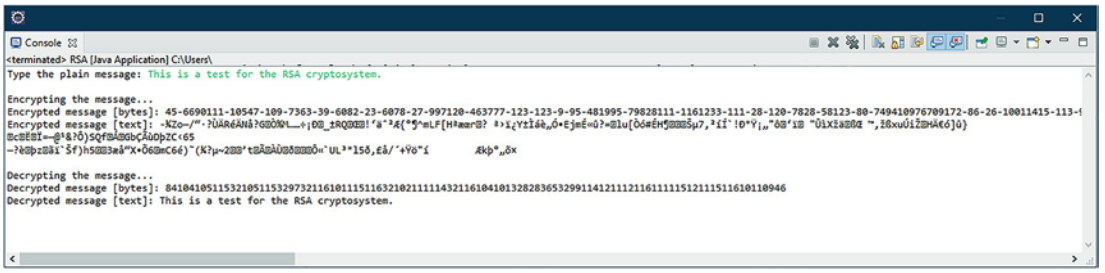

Console
<terminated> RSA [Java Application] C:\Users\
Type the plain message: This is a test for the RSA cryptosystem.
Encrypting the message...
Encrypted message [bytes]: 45-6690111-10547-109-7363-39-6082-23-6078-27-997120-463777-123-123-9-95-481995-79828111-1161233-111-28-120-7828-58123-80-749410976709172-86-26-10011415-113-
Encrypted message [text]: -ÑZo~/"·?ÙÄRéÄNå?GŒÒ¾L…÷¡Ø□_‡RQ□□□!'á^²Æ{°§^mLF[H#æœ□? ª›ï¿Y‡Ïáè„Ó•Ejœé«ù?=□lu[Òó#Éİ§□□□Šµ7,²íî`!Ð*Ÿ¡„"ðã'ïŒ "ÛìXŧæØŒ ™,ŽßxuÚiŽ□HÁ¢6]ù}
□c□ÊŒÏ=—@¹&?Ò)SQfŒ□GbÇÀÙÐþZC‹65
–?k□þz□ãi`Šf)h5□□3æå"X•Õ6□mC6é)¯(k?µ~2□□'t□Ã□ÀÙ□ß□□□Ô«¯UL³*15ð,£å/´+Ÿö"í Ækþ°„õ×
Decrypting the message...
Decrypted message [bytes]: 84104105115321051153297321161011151163210211114432116104101328283653299114121112116111151211151161011094 6
Decrypted message [text]: This is a test for the RSA cryptosystem.

Figure 11-1. *The output for the RSA cryptosystem*

As the implementation from Listing 11-1 uses `SecureRandom` and `BigIntegers`, it is not a naïve implementation, as it would be if simple data types (such as `int`) were used. However, the implementation of RSA can be even more advanced. Such an example can be found in [11], where the author uses `javax.crypto` and `X509EncodedKeySpec` to encode the public key. Regarding cryptanalysis, for example, a Pollard $p-1$ attack [13] or continued fraction attack [14] can be launched against the RSA encryption system. Because the public key is known by anyone, both attacks can find the private key if the values for d are small. A good implementation of these attacks is provided in [12].

Because of its construction, RSA is one of the most important public-key encryption systems. Based on this, a new hardness assumption was derived from the integer factorization, called the *RSA problem*, which is the foundation for subsequent cryptosystems. The RSA hardness assumption lies in the difficulty of computing the secret key by knowing the public key. Until this moment, there is no known efficient algorithm that solves this problem for a key size larger than 1,024 bits.

Another interesting feature of RSA is the homomorphic property of the multiplication operation. In the last decade or so, homomorphic encryption is a key branch in cryptography, because it allows (partially or totally, but limited) to make computations directly with the encrypted text, and the decryption of the result is the same as the result of the equivalent operations (via a homomorphism) applied on the plain text. Chapter 12 will discuss homomorphic encryption in more detail. For the RSA cryptosystem, the homomorphic operation is the following multiplication:

$$Enc(m_1) \bullet Enc(m_2) = m_1^e \bullet m_2^e \,(mod\ n) = (m_1 \bullet m_2)^e \,(mod\ n) = Enc(m_1 \bullet m_2)$$

Although RSA is a powerful cryptosystem, a weak implementation can lead to security issues. Further, we provide some best practices for the RSA implementation [3, 4].

- **Factoring problem.** The security of RSA is based on the difficulty of factoring large prime numbers; therefore, efficient algorithms for factoring could compromise RSA's security. Using sufficiently large keys is the primary defense against factoring attacks. As computational power increases, so too does the recommended key size. Currently, a minimum of 2,048 bits is recommended for RSA keys.
- **Timing attacks.** These are side-channel attacks where the attacker studies the time taken to execute cryptographic algorithms to infer about the key. To avoid this kind of leak, implementations can use constant-time algorithms to ensure that all operations take the same amount of time to execute, regardless of the input.
- **Chosen ciphertext attacks (CCA).** In these attacks, the attacker uses the public key to create ciphertexts and analyze responses when these are decrypted with the private key. To avoid CCAs, protocols such as Optimal Asymmetric Encryption Padding (OAEP) are used with RSA to provide security against these types of attacks.

ElGamal

Another notable example of public-key cryptosystems is ElGamal [5], whose theoretical foundations lie in the cyclic groups and the hardness assumption is the discrete logarithm problem (DLP). The specific DLPs for the ElGamal encryption system are the Diffie-Hellman assumption and the decisional Diffie-Hellman assumption (more technical details about DLP and Diffie-Hellman assumptions can be found in [7] and [8]). In this section, we present the ElGamal algorithms [5] and then show the implementation in Java.

The first algorithm is the *key generation algorithm*, and it works as follows:

1. Pick a prime number q and pick a cyclic group G (i.e., an element $g \in G$ should generate the entire group G) with the identity element $e \in G$.
2. Take an integer value from the interval $x \in \{1, \ldots, q-1\}$.
3. Compute the value $h = g^x$.

After running the key generation algorithm, the public key is $k_p = (G, q, g, h)$, and the private/secret key is $k_s = x$.

The second algorithm is the *encryption algorithm*, which is used to encrypt the plain message $m \in G$, and it is proceeds as follows:

1. Pick randomly a value $y \in \{1, \ldots q-1\}$.
2. Compute $s = h^y$.
3. Compute $c_1 = g^y$.
4. Compute $c_2 = m \bullet s$.

The encryption of the plain message m is the pair $c = (c_1, c_2)$.

The third algorithm is the decryption algorithm used to decrypt the encrypted message $c = (c_1, c_2)$ and to recover the plain message. It has the following steps:

1. Compute $s = c_1^x$.
2. Compute s^{-1}, the inverse of s.
3. Compute $m = c_2 \bullet s^{-1}$.

The cryptosystem is correct because the plain message is always recovered: $c_2 \bullet s^{-1} = (m \bullet s) \bullet s^{-1} = m$. An important technical aspect is that all operations are made within the group G; thus, all obtained results are elements of G.

The value s from the third step of the encryption algorithm is called the *shared secret*. This value is shared because, with some computations, it can be easily verified that it is the same as the value from the first step of the decryption $c_1^x = \left(g^y\right)^x = g^{xy} = h^y$. So, the sender and the receiver share the same value h^y.

The parameters of the system should be wisely chosen because, by taking a look at the encryption algorithm, we can see that an attacker knows $c = (c_1, c_2)$ and succeeds to recover m; then the shared secret can be computed as $c_2 \bullet m^{-1} = s$. In practice, to avoid this vulnerability, when a new message should be encrypted, a new value for y should be chosen. Listing 11-2 presents the Java implementation of the ElGamal cryptosystem.

Listing 11-2. ElGamal Implementation

```
1   package elgamal;
2
3   import java.io.BufferedReader;
4   import java.io.IOException;
5   import java.io.InputStreamReader;
6   import java.math.BigInteger;
7   import java.security.SecureRandom;
8   import java.nio.charset.StandardCharsets;
9
10  public class ElGamal {
11
12      private BigInteger prime_q, val_x, val_g, val_h;
13      private byte[] c1, c2;
14      private final BigInteger one = BigInteger.ONE;
15      private final BigInteger two = new BigInteger("2");
16      private final int maximumLength = 1024;
17      private SecureRandom random;
18
19      public void KeyGeneration() {
20          random = new SecureRandom();
21          prime_q = BigInteger.probablePrime(maximumLength, random);
22          do {
23              val_x = new BigInteger(maximumLength, random);
24          } while (val_x.compareTo(prime_q.subtract(one)) >= 0);
25
26          BigInteger p2 = prime_q.subtract(one).divide(two);
27
28          // take a generator g of the group
```

```
29          do {
30              val_g = new BigInteger(maximumLength, random).mod(prime_q);
31          } while (val_g.modPow(p2, prime_q).compareTo(one) == 0);
32
33          val_h = val_g.modPow(val_x, prime_q);
34      }
35
36      public void Encryption(byte[] plainMessage) {
37          BigInteger y, s;
38          do {
39              y = new BigInteger(maximumLength, random);
40          } while (y.compareTo(prime_q) >= 0);
41
42          s = val_h.modPow(y, prime_q);
43          c1 = val_g.modPow(y, prime_q).toByteArray();
44          c2 = new BigInteger(plainMessage).multiply(s).mod(prime_q).
            toByteArray();
45
46          System.out.println("Encrypted message [bytes]: " +
            byteToString(c1) + byteToString(c2));
47          System.out.println("Encrypted message [text]: " + new
            String(c1, StandardCharsets.UTF_8) + new String(c2,
            StandardCharsets.UTF_8));
48      }
49
50      public void Decryption() {
51          BigInteger s = new BigInteger(c1).modPow(val_x, prime_q);
52          BigInteger invS = s.modInverse(prime_q);
53          BigInteger m = invS.multiply(new BigInteger(c2)).mod(prime_q);
54
55          System.out.println("\nDecrypted message [bytes]: " +
            byteToString(m.toByteArray()));
56          System.out.println("Decrypted message [text]: " + new String
            (m.toByteArray(), StandardCharsets.UTF_8));
57      }
```

```
58
59      public static void main(String[] args) throws IOException {
60          ElGamal elGamal = new ElGamal();
61          elGamal.KeyGeneration();
62
63          try (BufferedReader d = new BufferedReader(new
            InputStreamReader(System.in))) {
64              System.out.print("Type the plain message: ");
65              String plainMessage = d.readLine();
66
67              System.out.println("\nEncrypting the message... ");
68              elGamal.Encryption(plainMessage.getBytes(StandardCharsets.
                UTF_8));
69              elGamal.Decryption();
70          }
71      }
72
73      private static String byteToString(byte[] byteArray) {
74          StringBuilder recoveredString = new StringBuilder();
75          for (byte byteVal : byteArray) {
76              recoveredString.append(Byte.toString(byteVal));
77          }
78          return recoveredString.toString();
79      }
80  }
```

Similarly, with RSA, the ElGamal implementation follows the algorithms described earlier. Figure 11-2 shows the result. Note that the values obtained in Figure 11-2 can be different every time because pseudorandom numbers are used. However, the plain message should be the same as the decrypted message every time. Examples of attacks that can be launched against the ElGamal cryptosystem are brute-force attacks, meet-in-the-middle attacks, and two-table attacks. A good reference for these attacks on ElGamal is [15].

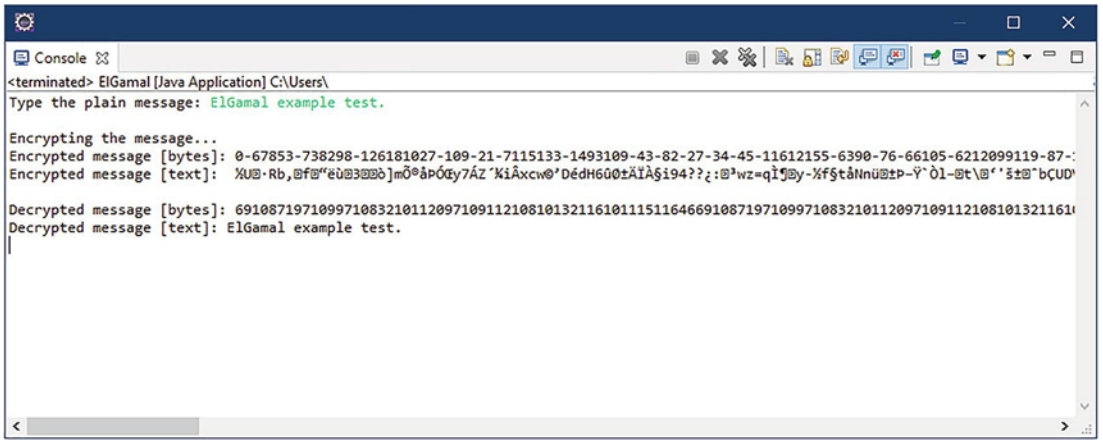

Console
<terminated> ElGamal [Java Application] C:\Users\
Type the plain message: ElGamal example test.
Encrypting the message...
Encrypted message [bytes]: 0-67853-738298-126181027-109-21-7115133-1493109-43-82-27-34-45-11612155-6390-76-66105-6212099119-87-
Encrypted message [text]:
Decrypted message [bytes]: 691087197109971083210112097109112108101321161011151164669108719710997108321011209710911210810132116
Decrypted message [text]: ElGamal example test.

Figure 11-2. *The result of the ElGamal implementation*

ElGamal is one of the best examples of an asymmetric encryption scheme. This cryptosystem also has homomorphic properties as follows:

$$Enc(m_1) \bullet Enc(m_2) = \left(g^{y_1}, m_1 \bullet h^{y_1}\right) \bullet \left(g^{y_2}, m_2 \bullet h^{y_2}\right) = \left(g^{y_1} \bullet g^{y_2},\ m_1 \bullet h^{y_1} \bullet m_2 \bullet h^{y_2}\right)$$
$$= \left(g^{y_1 y_2}, (m_1 m_2) \bullet h^{y_1 y_2}\right) = Enc(m_1 \bullet m_2)$$

Like RSA, some best practices for implementation should be considered for the ElGamal, as well. The following are presented some best practices [16–18]:

- **Discrete Logarithm Problem.** ElGamal security is based on the difficulty of solving the discrete logarithm problem. Faster algorithms for this problem could compromise the security of ElGamal; therefore, similar to RSA, increasing the key size for ElGamal can help protect against discrete logarithm advancements. Using groups where the discrete logarithm problem is harder (such as using elliptic curves) also enhances security.
- **Malleability.** ElGamal encryption is malleable, due to its homomorphic property, which in this case can be considered a security problem; attackers can alter a ciphertext's contents without needing to decrypt it, changing the message into another valid message. To avoid this, using digital signatures alongside encryption can help ensure integrity and authenticity, mitigating the risks posed by malleability.

- **Chosen Ciphertext Attacks.** Like RSA, ElGamal is also susceptible to chosen ciphertext attacks, especially in its basic form. Implementing ElGamal within a cryptosystem that includes message authentication or integrating it with protocols designed to resist CCA can safeguard against these attacks.

Merkle-Hellman

Another example of asymmetric encryption is the Merkle-Hellman cryptosystem [6]. Although it has been proven that it is insecure [10], it is interesting from the technical approach used, specifically the knapsack approach, presented in the next section.

The Knapsack Approach

In combinatorial optimization, the knapsack problem is a common problem. The input for the problem consists of a knapsack characterized by the maximum weight that it supports and a collection of objects characterized by the weight and profit of each object. It is requested to fill the knapsack with objects of the set without exceeding the maximum weight, but by obtaining a maximum profit. However, there are two versions of the problem: the discrete version (works only with whole objects) and the continuous version (may work with fractions of objects).

Technically, the knapsack problem can be formulated as follows: given a set of integers $A = \{a_1, \ldots a_n\}$ and a value S, the problem is to find the values $X = \{x_1, \ldots, x_n\}$, $x_i \in [0, 1]$, such that $S = a_1x_1 + \ldots + a_nx_n$. The work units in cryptography are bits; therefore, the possible values for x_i will become either 0 or 1.

Algorithms

Merkle-Hellman's cryptosystem has the general three algorithms: key generation, encryption, and decryption, which are presented in this section [9].

The *key generation algorithm* has the following steps:

1. Pick an integer value n as the dimension of the block.
2. Pick randomly n positive integer values $W = \{w_1, \ldots, w_n\}$ that fulfill the condition $w_k > \sum_{i=1}^{k-1} w_i, 1 < k \leq n$.
3. Pick randomly an integer q with $q > \sum_{i=1}^{n} w_i$.
4. Pick randomly an integer r with $\gcd(r, q) = 1$.
5. Calculate the elements of the tuple $B = (b_1, \ldots, b_n)$, where $b_i = rw_i \bmod q$.

The public key of the cryptosystem is $k_p = B$, and the secret key is $k_s = (W, q, r)$.

The *encryption algorithm* has one step, and the encrypted value c is obtained using the following formula:

$$c = \sum_{i=1}^{n} m_i b_i$$

where m_1 is the most significant bit and m_i is a bit of the message m with the length n.

The *decryption algorithm* has the following steps for recovering the plain message m from the encrypted value c:

1. Calculate the inverse r' of the value r, using the formula $r' = r^{-1} (mod\ q)$.
2. Calculate the value $c' = cr' (mod\ q)$.
3. Pick a subset $X' = \{x_1, \ldots x_k\}$ for which the elements fulfill the condition $c' = \sum_{i=1}^{k} w_i x_i$.
4. The plain message is recovered as $m = \sum_{i=1}^{k} 2^{n-x_i}$.

The third step of the decryption algorithm represents a problem called the *subset sum problem* for the value c'. This problem can be used by applying a greedy algorithm.

Advancements in Asymmetric Cryptography

Asymmetric cryptography has fast evolved to meet the increasing security demands of modern digital communication systems. Two of the most significant advancements in this field include the development and adoption of Elliptic Curve Cryptography (ECC) and the exploration into Post-Quantum Cryptography (PQC) [19–21]. Both of these advancements offer unique advantages over traditional public-key cryptography systems such as RSA and ElGamal, particularly in terms of security strength and computational efficiency.

Elliptic Curve Cryptography

Introduced in the mid-1980s, Elliptic Curve Cryptography (ECC) provides the same level of cryptographic strength as RSA but with much shorter keys. It is based on the algebraic structure of elliptic curves over finite fields. The difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP) ensures its security. The ECC has the following advantages [21]:

- **Efficiency.** ECC can achieve the same level of encryption strength as RSA with a significantly smaller key size, leading to faster processing and lower memory requirements. For instance, a 256-bit key in ECC is considered to provide comparable security to a 3072-bit key in RSA.
- **Bandwidth Savings.** The smaller key size also means that less data is transmitted during cryptographic operations, saving bandwidth and improving performance in high-volume or resource-constrained environments such as mobile devices or IoT applications.

ECC is widely used in many modern protocols, including TLS, SSL, and SSH. It is also the basis for many cryptographic standards used in domains like mobile telephony, banking, and government communications.

Post-Quantum Cryptography

As quantum computing advances, the ability of quantum algorithms (such as Shor's algorithm) to break traditional cryptographic systems like RSA and ECC has become an important concern. Post-Quantum Cryptography (PQC) refers to cryptographic algorithms believed to be secure against an attack by a quantum computer. The following are the latest developments in this area [19, 20]:

- **New Algorithms.** Several families of algorithms are being considered for PQC. These include lattice-based cryptography, hash-based signatures, code-based cryptography, and multivariate quadratic equations. Each of these families includes multiple candidates for specific cryptographic functions like encryption and digital signatures.
- **NIST Competition.** NIST is in the process of evaluating and standardizing post-quantum cryptographic algorithms. This ongoing effort aims to find the most secure and practical solutions for replacing current public-key cryptosystems.

The main advantage of PQC is its resistance to quantum attacks, which ensures the long-term security of data, even in a future where quantum computing will become part of our daily digital environments. Unlike traditional systems that often rely on a few classical hard problems, PQC offers a wide range of underlying mathematical approaches, potentially providing multiple layers of security through diversity. However, still remains one important challenge in adopting PQC with current technologies, namely, performance and storage. Many post-quantum algorithms currently have larger key sizes and require more processing power compared to classical systems. Optimizing these algorithms for practical use remains a key area of research.

While traditional asymmetric cryptographic methods like RSA and ElGamal have provided robust security for decades, advancements like ECC and PQC are essential to meet the evolving challenges in cybersecurity. These advancements not only offer improved efficiency and security but also prepare our cryptographic infrastructure for the advent of quantum computing.

Conclusion

The chapter explained what asymmetric encryption is and whether it is different from symmetric encryption. The three of the most important asymmetric encryption schemes were presented and then implemented: RSA, ElGamal, and Merkle-Hellman. Additionally, best practices for implementing these cryptosystems were presented. Lastly, the main advancements related to ECC and PQC were presented.

References

[1] Diffie, W., & Hellman, M. (1976). New directions in cryptography. IEEE Transactions on Information Theory, 22(6), 644–654.

[2] Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), 120–126.

[3] C. M. Kota and C. Aissi, "Implementation of the RSA algorithm and its cryptanalysis," in 2002 GSW, 2022. [Online]. Available: https://peer.asee.org/38785.pdf

[4] G. Shi, G. Wang, and D. Gu, "Further Cryptanalysis of a Type of RSA Variants," in Information Security, vol. 13640, W. Susilo, X. Chen, F. Guo, Y. Zhang, and R. Intan, Eds., in Lecture Notes in Computer Science, vol. 13640. Cham: Springer International Publishing, 2022, pp. 133–152. doi: 10.1007/978-3-031-22390-7_9.

[5] ElGamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory, 31(4), 469–472.

[6] Merkle, R., & Hellman, M. (1978). Hiding information and signatures in trapdoor knapsacks. IEEE Transactions on Information Theory, 24(5), 525–530.

[7] McCurley, K. S. (1990). The discrete logarithm problem. In Proc. of Symp. in Applied Math (Vol. 42, pp. 49–74).

[8] Bao, F., Deng, R. H., & Zhu, H. (2003, October). Variations of Diffie-Hellman problem. In International conference on information and communications security (pp. 301–312). Springer, Berlin, Heidelberg.

[9] Merkle, R., & Hellman, M. (1978). Hiding information and signatures in trapdoor knapsacks. IEEE Transactions on Information Theory, 24(5), 525–530.

[10] Shamir, A. (1982, November). A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem. In 23rd Annual Symposium on Foundations of Computer Science (SFCS 1982) (pp. 145–152). IEEE.

[11] RSA in Java, https://www.baeldung.com/java-rsa

[12] RSA-Encryption-Attacks, https://github.com/toknhawaiian/RSA-Encryption-Attacks

[13] Lydia, M. S., Budiman, M. A., & Rachmawati, D. (2020). On using Pollard's p-1 Algorithm to Factor RPrime RSA Modulus.

[14] Dujella, A. (2004). Continued fractions and RSA with small secret exponent. arXiv preprint cs/0402052.

[15] Allen, B. (2008). Implementing several attacks on plain ElGamal encryption. Iowa State University.

[16] M. El Laz, B. Grégoire, and T. Rezk, "Security analysis of ElGamal implementations," in SECRYPT 2020-17th International Conference on Security and Cryptography, SCITEPRESS-Science and Technology Publications, 2020, pp. 310–321. [Online]. Available: https://inria.hal.science/hal-03141511/document

[17] Y. Tsiounis and M. Yung, "On the security of ElGamal based encryption," in Public Key Cryptography, vol. 1431, H. Imai and Y. Zheng, Eds., in Lecture Notes in Computer Science, vol. 1431. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 117–134. doi: 10.1007/BFb0054019.

[18] T.-Y. Chang, M.-S. Hwang, and W.-P. Yang, "Cryptanalysis on an improved version of ElGamal-like public-key encryption scheme for encrypting large messages," Informatica, vol. 23, no. 4, pp. 537–562, 2012.

[19] D. S. Sakkari and M. M. Ulla, "Review on Insight into Elliptic Curve Cryptography," in Modern Approaches in Machine Learning & Cognitive Science: A Walkthrough, vol. 1027, V. K. Gunjan and J. M. Zurada, Eds., in Studies in Computational Intelligence, vol. 1027. Cham: Springer International Publishing, 2022, pp. 81–93. doi: 10.1007/978-3-030-96634-8_8.

[20] R. Renner and R. Wolf, "Quantum Advantage in Cryptography," AIAA Journal, vol. 61, no. 5, pp. 1895–1910, May 2023, doi: 10.2514/1.J062267.

[21] R. Bavdekar, E. J. Chopde, A. Bhatia, K. Tiwari, S. J. Daniel, and Atul, "Post Quantum Cryptography: Techniques, Challenges, Standardization, and Directions for Future Research." arXiv, Feb. 06, 2022. Accessed: Apr. 15, 2024. [Online]. Available: http://arxiv.org/abs/2202.02826

CHAPTER 12

Advanced Encryption Schemes

Advanced encryption schemes include fully homomorphic encryption (FHE) and searchable encryption (SE), which is a particular case of FHE. More than a decade ago when it was introduced, fully homomorphic encryption was considered a revolutionary approach in cryptography, although privacy homomorphism was introduced in the late 1970s [1]. Both types of encryption schemes are important for today's technologies as they correspond to the current cybersecurity needs. Both are mainly focused on technologies such as cloud computing.

Homomorphic Encryption

Homomorphic encryption is a type of encryption scheme that enables the user to make computations on encrypted data. This eliminates the need for decryption before applying some computations to the data. This saves a lot of time by eliminating additional procedures for downloading the data from the cloud server, decrypting it, applying computations on data, and then encrypting it back and storing it again on the cloud server in encrypted form. A classical use case of homomorphic encryption is when a user who is on a vacation in a foreign country wants to find out which is the nearest restaurant from their geographical position or where the interesting museums are. A simple search on the Internet for these places can reveal a lot of information about the user: what country they are in, their eating preferences (by looking at the restaurant type), what kind of knowledge they want to learn (by looking at the museum type), their current location, and so on. If the search engine system implemented a homomorphic encryption scheme, then this information about the user would not be accessible to the search engine system anymore, because all search operations would be made over

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_12

encrypted data and the user would receive the result in encrypted form. From this simple example, you can see the potential of homomorphic encryption for (almost) all domains of activities.

Let's suppose there are two structures of the same type (groups, rings, or fields): $(G_1, \perp)$, $(G_2, *)$. A function $f: G_1 \to G_2$ is called homomorphism between G_1, G_2 if

$$f(x_1 \perp x_2) = f(x_1) * f(x_2), \forall x_1, x_2 \in G_1$$

This relation says that the properties of the operation from G_1 are transferred to G_2 via a homomorphism *f*. In other words, in homomorphic encryption, all computations applied on the plaintext give the same result as applying the corresponding operations (via a homomorphism) over encrypted data. In this case, the result obtained from the plaintext is the same as the decrypted result obtained from the encrypted text. This is translated in technical aspects through the relationship that will be given next. In homomorphic encryption, there is an additional algorithm, besides the well-known key generation, encryption, and decryption. This algorithm is called *evaluation* (usually denoted with *Eval*), and it enables the evaluation of two encrypted texts c_1, c_2 (or an arbitrary number of encrypted text c_1, ...c_n) as follows:

$$Dec\left(k_{priv}, Eval_{k_{eval}}(c_1, c_2)\right) = f(m_1, m_2).$$

where m_1, m_2 are the corresponding plain messages from c_1 and c_2. Note that the evaluation algorithm uses an evaluation key, which is public.

From the four mathematical operations, only addition and multiplication should have homomorphic properties, because the *circuit* of any function can be expressed by using only the gates corresponding to these two operations. In short, functions are expressed through logical circuits that contain logical gates (AND, OR, XOR, NOT). More information about logical circuits and how the functions are expressed using them can be found in [2] and [3]. The problem with some homomorphic schemes is that they are constrained by physical limitations, as will be explained in the following categories; therefore, the operations will not be applied unlimitedly.

The intuition for homomorphic encryption is introduced in [1], in the form of privacy homomorphism. However, there are pros and cons to this approach; an example of danger is an attacker that alters the encrypted text, resulting in other plaintext that is not legitimate. The unpadded RSA [4] is considered among the first encryption schemes that support homomorphic properties and works as follows:

$$Enc(m_1) \bullet Enc(m_2) = m_1^e m_2^e \mod n = (m_1 m_2)^e \mod n = Enc(m_1 m_2),$$

where m_1, m_2 represents plain messages in the form for unpadded RSA and *Enc* is the encryption function of unpadded RSA.

Figure 12-1 shows the history of homomorphic encryption in a graphical form and represents some important encryption systems in each category.

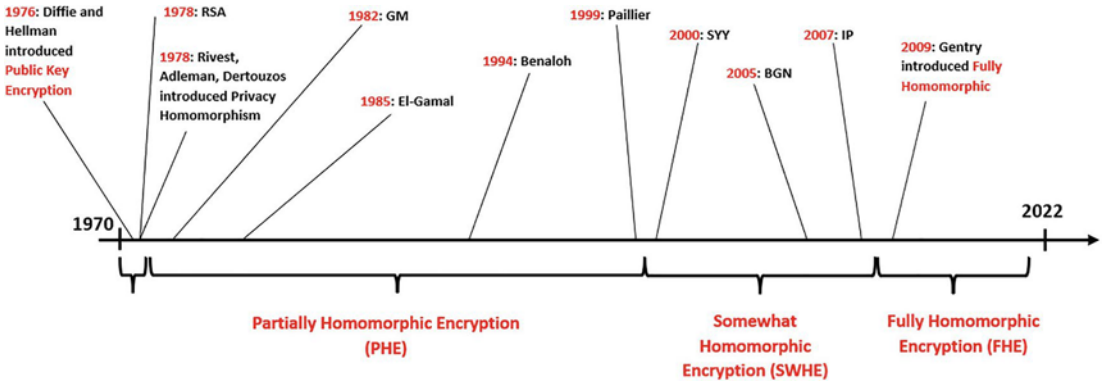


Figure 12-1. *The history of homomorphic encryption*

Homomorphic encryption is divided into three categories, as follows:

- **Partial homomorphic encryption (PHE).** For this category, one operation of the two (addition, multiplication) is homomorphic, but it can be applied an unlimited number of times. Well-known examples of PHE schemes are RSA [5], Goldwasser-Micali [6], and ElGamal [7]; [5] and [7] are implemented in Chapter 10.
- **Somewhat homomorphic encryption (SWHE).** This category includes encryption schemes for which both operations are homomorphic, but they can be used a limited number of times. An example of the SWHE scheme is [8].
- **Fully homomorphic encryption (FHE).** The last and the most exciting category of homomorphic encryption schemes is FHE, which enables both operations to be homomorphic and to be applied an unlimited number of times; therefore, any type of computation can be applied over encrypted data. In the research literature, FHE is considered the "holy grail of cryptography" or the "Swiss Army

knife of cryptography" [9]. Gentry's PhD thesis [10] introduced construction for the FHE schemes, using ideal lattices. This was a revolution in cryptography because it opened the door for the FHE. However, Gentry's scheme was very abstract and complex, and it was not practical. The complexity is in the mathematical tools that are used, namely, ideal lattices, and the unpractical aspect is given by the fact that with every evaluation a noise is introduced, and at some time this noise increases so much that the decryption is not possible anymore. FHE itself is divided into three generations: the original scheme, the bootstrapping schemes (that refresh the ciphertext at some point and eliminate the noise), and the schemes based on asymmetric multiplication for which $c_1 \bullet c_2 \neq c_2 \bullet c_1$, although both parts of inequality encrypt the product of the plain corresponding bits $b_1 \bullet b_2$.

Regarding homomorphic encryption, nowadays the focus is on FHE systems, because they can be quantum-resistant. This fact is due to their mathematical foundations, for example, lattices, ring-learning with errors, and NTRU, which are all resistant to Shor's algorithm. However, these mathematical tools are complex and exceed the limitations of physical devices nowadays; therefore, at the moment of writing this book, there is no practical implementation for an FHE scheme that works unlimitedly for any encrypted data. Attempts of FHE implementations exist and lead to powerful FHE libraries, but these are limited and cannot use the entire potential of the FHE schemes that they implement. The following are the FHE libraries, implemented in different programming languages:

- **HElib [11], [12].** This was developed by IBM and is one of the first implementations of FHE schemes; it implements the BGV scheme [13]. It has a good performance and is one of the few that implements the bootstrapping operation for the BGV scheme. HElib is written in C++.
- **SEAL [14].** This was developed by Microsoft and implements the BFV [15] and CKKS [16] encryption schemes. It can be used as a library in C# and C++.

- **TFHE [17].** This implements efficient bootstrap operations and enables functions to be applied directly to the encrypted data.
- **PALLISADE [18].** This library implements cryptographic primitives for ideal lattices.

For Java, there is a library that implements homomorphic encryption schemes that can be found at [21], but it is deprecated.

HE is standardized through the efforts of actors from different areas of business, such as industry, government, and academic institutions. The HomomorphicEncryption.org consortium, comprising industry, academia, and government stakeholders, has an important role in pushing forward the standardization of FHE. Their efforts have led to the publication of security standards and best practices for implementing FHE, aiming to harmonize the approaches used across different sectors and applications. Furthermore, NIST's Post-Quantum Cryptography Standardization process, while primarily focused on post-quantum algorithms, has indirectly influenced FHE research, highlighting the importance of quantum-resistant cryptographic solutions and encouraging the development of FHE schemes with quantum resistance in mind. All resources regarding HE standardization can be found at [19].

Recent advancements in FHE contains elements like computational overhead reduction, increased security, and standardization. A significant focus has been on reducing the computational overhead associated with FHE operations. Recent advancements include the development of more efficient bootstrapping techniques, which are crucial for refreshing ciphertexts to prevent noise buildup that makes the ciphertext indecipherable. For instance, innovations in *fast bootstrapping* [20–24] methods have drastically improved the time and computational resources required, making FHE more practical for real-world applications. Another area of progress is the optimization of underlying lattice-based cryptographic schemes. Techniques such as *sparse secret keys* [25, 26] and *ring switching* [27, 28] have contributed to faster encryption and decryption times while maintaining security integrity. Security enhancements in FHE schemes have been achieved through more robust mathematical constructs and improved implementation practices. For example, the adaptation of *module lattices* over traditional lattice approaches has offered stronger security assurances against both classical and quantum attacks. Ongoing research into the hardness assumptions underlying FHE schemes, such as the Learning With Errors (LWE) problem and its variants, has sustained the theoretical foundations, providing a clearer understanding of the security parameters needed to ensure robust encryption.

Searchable Encryption

Searchable encryption is a particular case of fully homomorphic encryption that allows search operations to be made over encrypted data. A classic example here is the medical unity that needs to retrieve a medical file for a specific patient from the servers by typing the personal Social Security number of the patient. Searchable encryption would work here as the search would be made directly on encrypted data; therefore, there is no need to retrieve the encrypted documents from the server or decrypt them. Searching for text based on a keyword is also an operation, and because the search is on encrypted data, then SE is a particular case of FHE.

The use-case scenario for SE is when there exists an entity that owns a set of documents $D = \{D_1, \ldots, D_n\}$, which are described by a set of keywords $K = \{kw_1, \ldots, kw_m\}$. The owner encrypts the documents using an SE scheme and stores them on a cloud server. Then, when the user wants a document (or more) that contains a specific keyword, the user will compute a special value called a *trapdoor* that the search process will be based on. The trapdoor value is then submitted to the cloud server that performs the search operation directly over the encrypted data and then sends the result to the user, who further decrypts the received documents.

Usually, searchable encryption schemes have two components: the *setup* component and the *retrieval* component. In the setup are operations such as data preprocessing, key generation, encryption, and storing data on the cloud servers, while the retrieval component contains operations such as query submission, search, and document retrieval. Searchable encryption has its specific types of users as follows:

- The *data owner* is the entity that owns the data, encrypts it, and stores it in encrypted form on the cloud server.
- The *data user* is the entity that works with the data. The data user can compute trapdoor values based on the query keyword and submit the queries to the cloud server. Also, the data user decrypts the encrypted result received from the server.
- The *cloud server* stores the encrypted data uploaded by the data owner and makes the search operation based on the trapdoor value received from the data user.

As you can see, the roles are very well established and delimited. In addition to the well-known key generation, encryption, and decryption algorithms, searchable encryption has two (or three) more algorithms based on which search is made. These are as follows:

- `BuildIndex`: This algorithm builds an index structure for the keywords that describe the documents stored on the server in the encrypted format. The algorithm is run by the data owner.
- `Trapdoor`: This algorithm computes a value called a *trapdoor* based on the query keyword chosen by the data user. Then, this value is submitted to the cloud server.
- `Search`: This algorithm searches through the encrypted data based on the trapdoor submitted by the data user. The search algorithm is performed by the cloud server.

Regarding their types, SE schemes can be symmetric searchable encryption (SSE) or public-key searchable encryption (PKSE). SSE, like all symmetric schemes, uses just one key for all operations, namely, the private key, while PKSE uses two types of keys, namely, the public key and the private key.

SE schemes should fulfill some specific security requirements, such as controlled searching, encrypted queries, and query isolation [29].

- *Controlled searching* means that the system does not accept search queries from unauthorized users. Moreover, the search operation made by the server should be applied directly to the encrypted data (without decrypting the data in any way).
- *Encrypted queries* means that the keyword for the search query should always be encrypted before being sent to the server.
- *Query isolation* means that the server learns nothing about the keyword, documents, the correspondence between a keyword and a set of documents, etc. In many studies, the server is described as "honest but curious," meaning that the server does its work as it should, but it can try to find relevant information in the process. Query isolation is also required in this setup for the server.

Among the first SE schemes were [30] and [31], and they are important for the categories in which they are included (SSE and PKSE, respectively) because they are considered primary works.

Two libraries that implement searchable encryption schemes are [32] and [33].

Recent advancements in SE include efficiency, practicality, and enhanced security models. Innovations in SE have aimed at making it more efficient and practical for large-scale applications. This includes the development of *multi-keyword searchable encryption* schemes [34–36] that allow for more complex queries over encrypted data, significantly enhancing the usability of SE in real-world databases and cloud storage solutions. Breakthroughs in structured encryption have enabled SE schemes to support not only keyword searches but also range queries, wildcard searches, and other complex database operations, making SE increasingly usable for commercial and enterprise environments. Also, recent advancements have focused on achieving stronger security guarantees, such as *forward privacy* (ensuring that newly added documents do not compromise the security of previous searches) and *backward privacy* (protecting the search history even if documents are later removed) [37–39].

Efforts to formalize the security properties of SE schemes have led to a better understanding of the trade-offs between search functionality and privacy, guiding the design of new schemes that offer improved functionalities. While SE does not yet have a standardized framework compared to FHE, there has been a movement toward establishing best practices and protocols, especially in industry applications. Organizations such as the Cloud Security Alliance[1] have begun to outline guidelines for secure cloud data searchability, which implicitly incorporate SE principles. Protocols for verifiable search operations, ensuring that the results returned from a search on encrypted data are correct and complete, have gained traction. These protocols are vital for trust in SE applications, particularly in sensitive sectors like healthcare and finance.

Conclusion

This chapter presented two of the most important topics of cryptography for today's research community. Fully homomorphic encryption is important because the techniques that it uses and its mathematical properties make it quantum resistant. This is a powerful property as quantum computers already exist in dedicated research

[1] https://cloudsecurityalliance.org/

laboratories. Searchable encryption is also an important topic, being a particular case of fully homomorphic encryption. Its focus is on searching for specific data that fulfills some criteria, an aspect that is very useful in the database field.

References

[1] Rivest, R. L., Adleman, L., & Dertouzos, M. L. (1978). On data banks and privacy homomorphisms. Foundations of secure computation, 4(11), pp. 169–180.

[2] Basic Gates and Functions. Available online: http://www.ee.surrey.ac.uk/Projects/CAL/digital-logic/gatesfunc/index.html

[3] CS 370 Computer Architecture Spring 2020. Available online: https://taoxie.sdsu.edu/cs370/

[4] Rivest, R. L., Shamir, A., Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), pp. 120–126.

[5] Ronald L. Rivest, Adi Shamir, and Leonard Adleman, "A method for obtaining digital signatures and public-key cryptosystems." Communications of the ACM 21.2 (1978): 120–126.

[6] Shafi Goldwasser and Silvio Micali, "Probabilistic encryption and how to play mental poker keeping secret all partial information." Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing. 1982.

[7] Taher ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms." IEEE transactions on information theory 31.4 (1985): 469–472.

[8] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim, "Evaluating 2-DNF formulas on ciphertexts." Theory of Cryptography Conference. Springer, Berlin, Heidelberg, 2005.

[9] B. Barak and Z. Brakerski. "The Swiss Army Knife of Cryptography," http://windowsontheory.org/2012/05/01/theswiss-army-knife-of-cryptography/, 2012.

[10] Craig Gentry, "Fully homomorphic encryption using ideal lattices." Proceedings of the forty-first annual ACM symposium on Theory of computing. 2009.

[11] Halevi, S., Shoup, V. (2013). Design and implementation of a homomorphic encryption library. IBM Research (Manuscript), Vol. 6, pp. 12–15.

[12] HElib Documentation, https://homenc.github.io/HElib/

[13] Brakerski, Z., Gentry, C., & Vaikuntanathan, V. (2014). (Leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT), 6(3), pp. 1–36.

[14] Microsoft SEAL (release 3.2.0). February 2019, http://sealcrypto.org. Microsoft Research, Redmond, WA

[15] Fan, J., & Vercauteren, F. (2012). Somewhat Practical Fully Homomorphic Encryption. IACR Cryptology ePrint Archive, Report 2012/144.

[16] Cheon, J. H., Kim, A., Kim, M., & Song, Y. (2017i). Homomorphic encryption for arithmetic of approximate numbers. In International Conference on the Theory and Application of Cryptology and Information Security (pp. 409–437). Springer, Cham.

[17] Chillotti, I., Gama, N., Georgieva, M., & Izabachene, M. (2016). Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In international conference on the theory and application of cryptology and information security (pp. 3–33). Springer, Berlin, Heidelberg.

[18] Polyakov, Y., Rohloff, K., Ryan, G. W. (2017). PALISADE lattice cryptography library user manual. Cybersecurity Research Center, New Jersey Institute of Technology (NJIT), Tech. Rep. https://git.njit.edu/groups/palisade

[19] Albrecht M, Chase M, Chen H, Ding J, Goldwasser S, Gorbunov S, Halevi S, Hoffstein J, Laine K, Lauter K, Lokam S, Micciancio D, Moody D, Morrison T, Sahai A, Vaikuntanathan V. (2018). Homomorphic encryption security standard. Technical report. Toronto, Canada: HomomorphicEncryption.org

[20] H. V. L. Pereira, 'Bootstrapping Fully Homomorphic Encryption over the Integers in Less than One Second', in Public-Key Cryptography – PKC 2021, vol. 12710, J. A. Garay, Ed., in Lecture Notes in Computer Science, vol. 12710. , Cham: Springer International Publishing, 2021, pp. 331–359. doi: 10.1007/978-3-030-75245-3_13.

[21] A. Kim et al., 'General bootstrapping approach for RLWE-based homomorphic encryption', IEEE Transactions on Computers, 2023, [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10261340/

[22] Y. Lee et al., 'Efficient FHEW Bootstrapping with Small Evaluation Keys, and Applications to Threshold Homomorphic Encryption', in Advances in Cryptology – EUROCRYPT 2023, vol. 14006, C. Hazay and M. Stam, Eds., in Lecture Notes in Computer Science, vol. 14006. , Cham: Springer Nature Switzerland, 2023, pp. 227–256. doi: 10.1007/978-3-031-30620-4_8.

[23] R. Geelen, I. Iliashenko, J. Kang, and F. Vercauteren, 'On Polynomial Functions Modulo $$p^e$$ and Faster Bootstrapping for Homomorphic Encryption', in Advances in Cryptology – EUROCRYPT 2023, vol. 14006, C. Hazay and M. Stam, Eds., in Lecture Notes in Computer Science, vol. 14006. , Cham: Springer Nature Switzerland, 2023, pp. 257–286. doi: 10.1007/978-3-031-30620-4_9.

[24] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, 'Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs', IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 114–148, 2021.

[25] A. Nawaz et al., 'Secure Distributed Sparse Gaussian Process Models Using Multi-Key Homomorphic Encryption', in Proceedings of the AAAI Conference on Artificial Intelligence, 2024, pp. 14431–14439. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/29357

[26] J.-P. Bossuat, J. Troncoso-Pastoriza, and J.-P. Hubaux, 'Bootstrapping for Approximate Homomorphic Encryption with Negligible Failure-Probability by Using Sparse-Secret Encapsulation', in Applied Cryptography and Network Security, vol. 13269, G. Ateniese and D. Venturi, Eds., in Lecture Notes in Computer Science, vol. 13269. , Cham: Springer International Publishing, 2022, pp. 521–541. doi: 10.1007/978-3-031-09234-3_26.

[27] S. Park, W. Song, S. Nam, H. Kim, J. Shin, and J. Lee, 'HEaaN.MLIR: An Optimizing Compiler for Fast Ring-Based Homomorphic Encryption', Proc. ACM Program. Lang., vol. 7, no. PLDI, pp. 196–220, Jun. 2023, doi: 10.1145/3591228.

[28] A. Kim et al., 'General bootstrapping approach for RLWE-based homomorphic encryption', IEEE Transactions on Computers, 2023, [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10261340/

[29] Song, D. X., Wagner, D., Perrig, A. (2000). Practical techniques for searches on encrypted data. In Proceeding 2000 IEEE Symposium on Security and Privacy. SP 2000 (pp. 44–55). IEEE.

[30] E. J. Goh, "Secure indexes." IACR Cryptology ePrint Archive, 2003, 216

[31] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search." In International conference on the theory and applications of cryptographic techniques (pp. 506–522). 2004. Springer, Berlin, Heidelberg

[32] encryptedsystems/Clusion, https://github.com/encryptedsystems/Clusion

[33] Crypteron, https://www.crypteron.com/

[34] D. Sangeetha, S. S. Chakkaravarthy, S. C. Satapathy, V. Vaidehi, and M. V. Cruz, 'Multi keyword searchable attribute based encryption for efficient retrieval of health Records in Cloud', Multimed Tools Appl, vol. 81, no. 16, pp. 22065–22085, Jul. 2022, doi: 10.1007/s11042-021-10817-z.

[35] Y. Zhang, T. Zhu, R. Guo, S. Xu, H. Cui, and J. Cao, 'Multi-keyword searchable and verifiable attribute-based encryption over cloud data', IEEE Transactions on Cloud Computing, vol. 11, no. 1, pp. 971–983, 2021.

[36] Q. Tong, Y. Miao, J. Weng, X. Liu, K.-K. R. Choo, and R. H. Deng, 'Verifiable fuzzy multi-keyword search over encrypted data with adaptive security', IEEE Transactions on Knowledge and Data Engineering, vol. 35, no. 5, pp. 5386–5399, 2022.

[37] Z. Wu and R. Li, 'OBI: a multi-path oblivious RAM for forward-and-backward-secure searchable encryption', in NDSS, 2023. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2023-423-paper.pdf

[38] L. Chen, J. Li, and J. Li, 'Towards forward and backward private dynamic searchable symmetric encryption supporting data deduplication and conjunctive queries', IEEE Internet of Things Journal, 2023, [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10121755/

[39] T. Chen, P. Xu, W. Wang, Y. Zheng, W. Susilo, and H. Jin, 'Bestie: Very Practical Searchable Encryption with Forward and Backward Security', in Computer Security – ESORICS 2021, vol. 12973, E. Bertino, H. Shulman, and M. Waidner, Eds., in Lecture Notes in Computer Science, vol. 12973. , Cham: Springer International Publishing, 2021, pp. 3–23. doi: 10.1007/978-3-030-88428-4_1.

CHAPTER 13

Identification Schemes

Identification schemes are one of the most important types of scheme used in cryptography. There are two entities, namely, the prover and the verifier, in a scheme. The prover tries to prove its identity to the verifier in such a way that the prover cannot be impersonated by an adversary either in the process of demonstrating its identity or after the demonstration. Once the identity is confirmed, not even the verifier can alter the prover's identity. The process of proving the identity may end with acceptance or rejection (termination without acceptance).

A rudimentary technique of identification is using a password in a particular process. This method provides weak authentication. Identification schemes have many applications, such as access control. Access control is an important part of a system because it manages the access to different resources based on the privileges of each type of user within the system. However, the identification schemes are not related only to human users; they can be used for a specific process authentication. For example, a process/application may use a specific port for communication transmission, or some automated task within an application is implemented to do specific operations according to its privileges. Another handy example is related to our daily lives: we use our biometric features (fingerprints, face) to unlock our smartphones. The following are some of the most used scenarios for including identification schemes [1, 2]:

Digital Banking

- **Online Banking Authentication:** Financial institutions use advanced identification schemes for online banking services to protect users' accounts from unauthorized access. For example, multifactor authentication (MFA) requires customers to provide something they know (a password), something they have (a mobile device), and sometimes something they are (biometric data) to access their accounts. This layered system reduces the risk of fraud and identity theft.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_13

- **Transaction Signing:** To authorize high-value transactions, some banks have implemented digital signature schemes that use cryptographic keys stored on secure devices (like smart cards or mobile phones with secure elements). Customers use these to digitally sign transactions, providing nonrepudiation and integrity.

1. *Secure Online Voting*

   - **Voter Authentication.** For online voting systems, identification schemes ensure that each vote is cast by a legitimate, eligible voter and that voters cannot vote more than once. A zero-knowledge proof identification scheme could allow voters to prove their eligibility without revealing their identity, maintaining both security and voter anonymity.

   - **Vote Integrity and Secrecy.** Cryptographic digital signatures ensure that each vote is both unaltered and authenticated, without linking the vote back to the voter, preserving the secrecy of the ballot.

2. *IoT Device Authentication*

   - **Smart Home Security.** In smart home ecosystems, devices like smart locks and security cameras use cryptographic identification schemes to authenticate each other. This ensures that commands are accepted only from authorized devices or users, increasing security against unauthorized access or spoofing attacks.

   - **Supply Chain and Industrial IoT.** Identification schemes authenticate IoT devices in supply chains or industrial settings, ensuring data integrity and secure communication. For example, a smart sensor in a supply chain might use a cryptographic scheme to sign its data, ensuring that the data has not been altered from the point of origin to the destination.

3. *Blockchain Identities*

    - **Decentralized Identities (DIDs).** Blockchain technology enables the creation of decentralized identities, where individuals or devices can own and control their identity without reliance on a central authority. Using public-key cryptography, users can prove ownership of their identity through digital signatures without exposing sensitive personal information.

An identification protocol takes place in real time, showing that the prover is active and operational during the whole process of identification. To identify a prover, the following techniques can be used [3–5]:

- **Something known.** In this category, passwords, personal identification numbers (PINs), or a secret key can be included.
- **Something possessed.** This category usually includes physical devices such as magnetic-stripe cards, smart cards, password generator devices for one-time passwords, etc.
- **Something inherited.** This category is related to human physical characteristics, namely, biometric features, such as fingerprint, handwriting or handwritten signatures, voice, etc.

Weak authentication is made by using passwords, and there are several approaches. In this case, the passwords can be stored in specific files that are protected from reading and writing. However, the passwords should not be stored in plaintext. Instead, the password should be hashed using a particular hash function or hashing technique (see Chapter 9 for more about hash functions). Another approach involves defining rules for passwords, for example using a minimum number of characters, uppercase letters, special symbols, etc. These rules have the purpose of protecting against dictionary attacks, which can easily break predictable passwords. For an additional level of security, the passwords can be, for example, XORed with a specific value and then on the obtained value is applied a hash function. Finally, the hashed value is stored for further authentication [3–5].

The identification schemes that are based on passwords are exposed to the following types of attacks: replay of fixed passwords, exhaustive password search, password-guessing and dictionary attacks, etc.

There are two other techniques similar to passwords: PINs and two-step authentication. One-time passwords (OTPs) are an intermediary technique between weak authentication (passwords) and strong authentication (zero-knowledge proofs); they provide a stronger authentication than passwords. As the name suggests, when the user wants to authenticate, a different password is used each time. For the OTP, there are several ways in which they can be used: a list of OTPs that is shared between the authentication server and the user, OTPs that are updated continuously, and OTPs that use hash functions [3–5].

Zero-knowledge proofs are the most secure way to prove an identity, because they provide strong authentication. To identify itself, the prover may use zero-knowledge proofs, and usually, the proof consists of several pairs of challenge-response messages exchanged between parties (prover and verifier). To prove the identity, the verifier challenges the prover, and the prover must respond with values known only by itself that prove its identity. It works like a secret key-public key pair: the prover generates its response based on its secret key, and the verifier checks the response using the prover's public key. When the messages are exchanged, the verifier learns nothing additional about the verifier (has "zero knowledge"), and no additional information about the prover is leaked such that the verifier convinces other entities about the prover's identity [3–5]. Figure 13-1 shows a general form of message exchange.

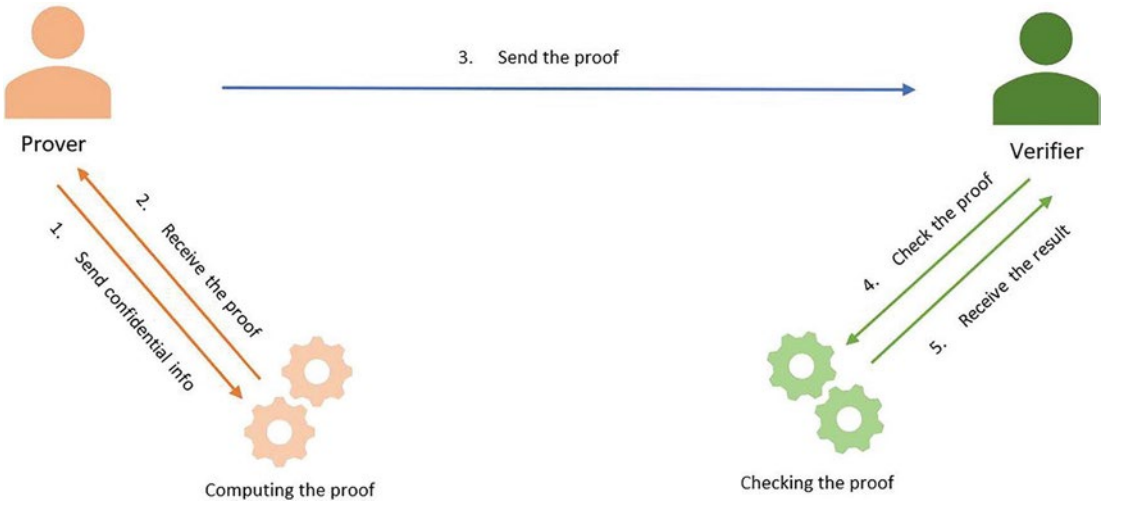


Figure 13-1. *Exchanging the messages between the prover and verifier*

The identification schemes that are based on zero-knowledge proofs have the following characteristics [3–5]:

- **Completeness.** The message response of the prover should be marked as valid when the claim "prover tries to convince the verifier" is true.
- **Soundness.** The message response of the prover should be marked as valid with a very small probability when the claim "prover tries to convince the verifier" is false.
- **Zero-knowledge.** The verifier does not learn anything additional about the prover in the process of identification.

FFS Identification Protocol

An example of a zero-knowledge identification scheme is the Feige-Fiat-Shamir (FFS) identification protocol [6], in which the prover tries to prove its identity to the verifier in several t executions. The protocol has four stages: establishing the system parameters, generating keys for provers, exchanging messages, and taking actions. The protocol FFS is as follows [6]:

- **System Parameters.** The trusted authority (TA) chooses two prime numbers p and q, such that p and q are congruent to 3 modulo 4. The two values are kept secret, and the value $n = pq$ is computed and made publicly available. Note that the problem of integer factorization for n should be hard. Considering the probability of impersonation based on the best attack as being 2^{-kt}, then the security parameters for the system are k and t, where t also gives the number of executions.
- **Keys Generation.** Every prover should follow these steps:

  From the interval $[1, n-1]$, choose randomly k integer values, and call them $s_1, \ldots, s_k$. Then choose randomly k bit values $b_1, \ldots b_k$. Calculate the following value for each $1 \le i \le k$: $v_i = (-1)^{b_i} \bullet \left(s_i^2\right)^{-1} \ mod \ n$.

  The public key for the prover is $PK = (v_1, \ldots, v_k)$, and the secret key is $SK = (s_1, \ldots, s_k)$.

- **Messages Exchange.** For each round *t*, three messages are exchanged between the prover and the verifier (see the next step for the formulas for each value here):

  Prover to verifier: Send the value $x = \pm r^2 \ mod \ n$.

  Verifier to prover: Send the tuple $e = (e_1, \ldots, e_k)$, $e_i \in \{0, 1\}$.

  Prover to verifier: Send the value $y = r \bullet \prod_{e_j=1} s_j^{e_j} \ mod \ n$.

- **Actions.** For each round *t*, for the previous values *x*, *e*, *y* some computations should be made. The identity of the prover is accepted if all *t* rounds terminate with success; otherwise, the identity is rejected. For *x*, *e*, *y*, the computations are as follows:

  The prover generates randomly an integer value *r*, with $1 \leq r \leq n - 1$. Then, the prover generates randomly a bit value *b* and calculates $x = (-1)^b \bullet r^2 \ mod \ n$. The value *x* is called the *witness*, and it is sent to the verifier.

  The verifier generates randomly *k* bit values in the form of $e = (e_1, \ldots, e_k)$ and sends them to the prover. The values from the tuple are called the *challenge*.

  The prover calculates the value $y = r \bullet \prod_{e_j=1} s_j^{e_j} \ mod \ n$ and sends it to the verifier. This value is called the *response*.

  The verifier calculates the value $z = y^2 \bullet \prod_{j=1}^{n} v_j^{e_j} \ mod \ n$. Then it checks whether $z = \pm x$ and $z \neq 0$.

Next, the implementation in Java is presented in Listing 13-1, and the result can be checked in Figure 13-2.

Listing 13-1. *The Implementation of the FFS Identity Scheme*

```
1   import java.math.BigInteger;
2   import java.util.ArrayList;
3   import java.util.BitSet;
4   import java.util.List;
5   import java.util.Random;
```

```
6   import java.util.Scanner;
7
8   public class FFSScheme {
9
10      // declare system parameters: prime numbers p,q, computed n,
           and given k
11      private static BigInteger p;
12      private static BigInteger q;
13      private static BigInteger n;
14      private static int k;
15
16      // used to generate random values
17      private static Random random = new Random();
18
19      // the set of k random integer values (private/secret key)
20      private static List<BigInteger> random_s_set = new ArrayList<>();
21      // the set of k random bits
22      private static BitSet random_b_set;
23      // the set of k computed values in the form of tuple v (public key)
24      private static List<BigInteger> computed_v_set = new ArrayList<>();
25
26      // the witness
27      private static BigInteger x;
28      // the set of the challenge values
29      private static BitSet random_e_set;
30      // the response
31      private static BigInteger y;
32
33      // the product used in the computation of y
34      private static BigInteger product_s;
35      // the product used in the computation of z
36      private static BigInteger product_v;
37
38      // the random integer value used in messages exchange
39      private static BigInteger r;
40
```

```
41      // the random bit value used in messages exchange
42      private static int b;
43
44      // method that implements system parameters generation
45      private static void GenerateParameters() {
46          // p = generate a big prime integer on 128 bits
47          p = BigInteger.probablePrime(128, random);
48          // q = generate a big prime integer on 128 bits
49          q = BigInteger.probablePrime(128, random);
50          // n = pq
51          n = p.multiply(q);
52
53          // k is given by the user
54          System.out.println("k= ");
55          try (Scanner sc = new Scanner(System.in)) {
56              k = sc.nextInt();
57          }
58          random_b_set = new BitSet(k);
59          random_e_set = new BitSet(k);
60      }
61
62      private static void GenerateKeys() {
63          for (int i = 0; i < k; i++) {
64              // generate each s_i value
65              // the number of bits used for representation of each s_i
66              // should be the same as n
67              BigInteger s_i = new BigInteger(n.bitLength() + 1,
                random).mod(n);
68
69              // check whether gcd(s_i, n) = 1
70              while (!s_i.gcd(n).equals(BigInteger.ONE)) {
71                  s_i = new BigInteger(n.bitLength() + 1, random).mod(n);
72              }
73
74              // add the value s_i for which gcd(s_i, n) = 1 to the s set
75              random_s_set.add(s_i);
```

```
76
77              // generate random bit b_i and add it to the set of bits b
78              random_b_set.set(i, random.nextBoolean());
79
80              // compute (-1)^(b_i)
81              BigInteger minus_one_pow = (((new BigInteger("-1")).
                pow(random_b_set.get(i) ? 1 : 0)));
82
83              // compute v_i = (-1)^(b_i) x ((s_i)^2)^(-1) mod n
84              BigInteger computed_v_i = minus_one_pow.multiply
                (random_s_set.get(i).pow(2)).modInverse(n);
85
86              // add v_i to the set v
87              computed_v_set.add(computed_v_i);
88          }
89      }
90
91      private static void ExchangeMessages() {
92          // generate random integer r
93          // mod n ensures that r is in [1, n-1]
94          r = new BigInteger(n.bitLength() + 1, random).mod(n);
95
96          // generate random bit b
97          b = random.nextBoolean() ? 1 : 0;
98
99          // compute x = (-1)^b x r^2
100          x = (((new BigInteger("-1")).pow(b)).multiply((r.pow(2))))
             .mod(n);
101
102         // generate random bit e_i and add it to the set of
               challenge bits e
103         for (int i = 0; i < k; i++) {
104             random_e_set.set(i, random.nextBoolean());
105         }
106
```

```
107         // compute the product (s_1)^(e_1) x ... x (s_k)^(e_k)
108         product_s = BigInteger.ONE;
109         for (int i = 0; i < k; i++) {
110             product_s = product_s.multiply(random_s_set.get(i).
            pow(random_e_set.get(i) ? 1 : 0));
111         }
112
113         // compute y = [the product from above] x r (mod n)
114         y = product_s.multiply(r.mod(n)).mod(n);
115     }
116
117     private static void CheckProof() {
118         // compute the product (v_1)^(e_1) x ... x (v_k)^(e_k)
119         product_v = BigInteger.ONE;
120         for (int i = 0; i < k; i++) {
121             product_v = product_v.multiply(computed_v_set.get(i).
            pow(random_e_set.get(i) ? 1 : 0));
122         }
123
124         //compute z = [the product from above] x y^2 (mod n)
125         //BigInteger z = (y.pow(2).mod(n)).multiply(product_v).mod(n);
126
127         BigInteger z = new BigInteger("1234555");
128
129         System.out.print("\nz = " + z.toString());
130
131         System.out.print("\nResponse: ");
132
133         //check whether z = +/- x and z != 0
134         if ((z.equals(x) || z.equals(x.negate().mod(n))) &&
            !z.equals(new BigInteger("0"))) {
135             System.out.print("\nAccept");
136         } else {
137             System.out.print("\nReject");
138         }
139     }
228
```

```
140
141     public static void main(String[] args) {
142
143         GenerateParameters();
144
145         GenerateKeys();
146
147         ExchangeMessages();
148
149         System.out.println("\n=== System parameters and computed
            values ===");
150
151         System.out.print("\np = " + p);
152         System.out.print("\nq = " + q);
153         System.out.print("\nn = " + n);
154
155         System.out.print("\nSecret key: ( ");
156         //for (int i = 0; i < k; i++)
157         //    System.out.print(random_s_set.get(i) + " ");
158         random_s_set.forEach(s -> System.out.print(s + " "));
159         System.out.print(")");
160
161         System.out.print("\nPublic key: ( ");
162         //for (int i = 0; i < k; i++)
163         //    System.out.print(computed_v_set.get(i) + " ");
164         computed_v_set.forEach(v -> System.out.print(v + " "));
165         System.out.print(")");
166
167         System.out.print("\nr = " + r);
168
169         System.out.print("\nb = " + b);
170
171         System.out.print("\nx = " + x);
172
173         System.out.print("\nRandom bits e: ( ");
```

```
174         for (int i = 0; i < k; i++)
175             System.out.print(random_e_set.get(i) ? 1 + " " : 0 + " ");
176         System.out.print(")");
177
178         System.out.print("\ny = " + y);
179
180         CheckProof();
181     }
182 }
```

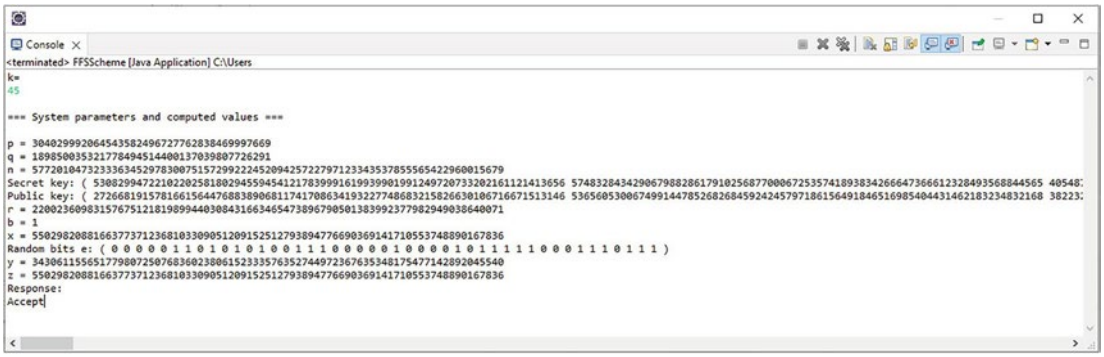


Figure 13-2. *The result for the FFS identification scheme*

The code from Listing 13-1 contains on the first six lines the imported libraries used in the program. The code is divided into four methods, namely, the method that implements the generation of the system parameters, the method for the keys generation, the method that computes the values exchanged between the prover and verifier, and the method that checks whether the response checks the identity of the prover or not. Each method follows the computations presented in the technical details of the FFS scheme. To check if the verification fails when a wrong value is sent, just comment line 132 and uncomment line 134. It is obvious that the integer from line 134 does not fulfill the conditions $z = \pm x$ and $z \neq 0$. Indeed, in this case, the identity is rejected (see Figure 13-3).

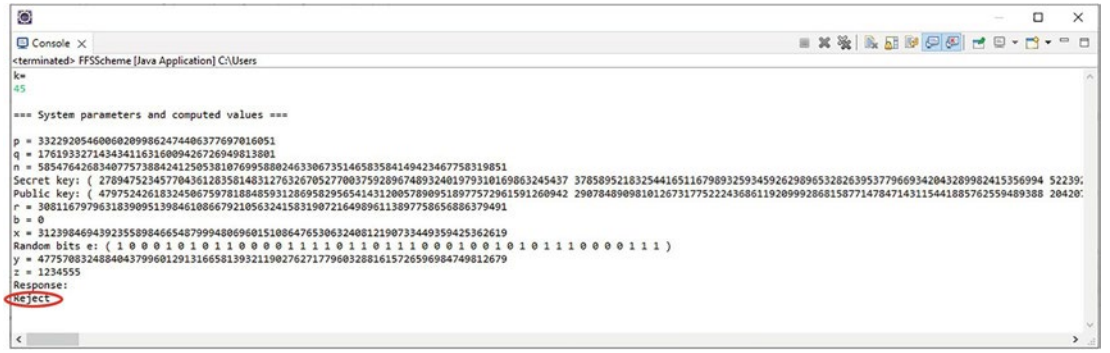


Figure 13-3. *Rejecting the identity in the FFS scheme*

Note that the code from Listing 13-1 is implemented for the $t = 1$ round. For an arbitrary number of rounds, the `ExchangeMessages` and `CheckProof` should be called together for that number of times. If each time the message is "Accept," then the identity of the prover is confirmed; otherwise, the identity of the prover is not confirmed.

The FFS identification protocol has the following security properties [3]:

- It is provably secure against the chosen message attacks.
- The hardness assumption lies in the integer factorization problem.
- The verifier learns nothing about the prover in the process of message exchange.
- Well-chosen system parameters give fewer chances of impersonation. For example, choosing the values k, t such that $kt = 20$, the possibility of impersonation is 1 in a million.
- However, if kt is constant while k is increased and t is decreased, it makes the identity scheme not zero-knowledge anymore.

Conclusion

This chapter showed that there are several types of identification schemes that confirm the identity of an entity, from ones that provide weak authentication (such as passwords) to ones that provide strong authentication (zero-knowledge proofs).

One of the most popular identification protocols is Feige-Fiat-Shamir (FFS), whose hardness assumption lies on the integer factorization problem. This chapter showed how to implement the FFS protocol.

References

[1] K. Kurosawa and S.-H. Heng, 'The Power of Identification Schemes,' in Public Key Cryptography - PKC 2006, vol. 3958, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., in Lecture Notes in Computer Science, vol. 3958. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–377. doi: 10.1007/11745853_24.

[2] A. Frankó, G. Vida, and P. Varga, "Reliable identification schemes for asset and production tracking in industry 4.0," Sensors, vol. 20, no. 13, p. 3709, 2020.

[3] Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A. (2018). Handbook of applied cryptography. CRC Press.

[4] Stinson, D. R. (2005). Cryptography: theory and practice. Chapman and Hall/CRC.

[5] Kurosawa, K., & Heng, S. H. (2006, April). The power of identification schemes. In International Workshop on Public Key Cryptography (pp. 364–377). Springer, Berlin, Heidelberg.

[6] Feige, U., Fiat, A., & Shamir, A. (1988). Zero-knowledge proofs of identity. Journal of cryptology, 1(2), 77–94.

CHAPTER 14

Signature Schemes

In the real world, when a message needs to be authenticated by an individual, it is written on paper, and then the individual signs the message. Therefore, it is known that this specific message belongs to a specific person. In the digital world, things can happen similarly: a piece of data (message) is created by an individual, and then the message is signed digitally with a specific signature by that individual. Digital signatures are an important component of modern cryptography and cybersecurity, providing a way to verify the authenticity and integrity of digital messages or documents. They have an important role in different applications, from email security to software distribution and beyond.

Digital signature schemes are public-key schemes that authenticate a message and check its integrity. The receiver and any third party may check individually the bond between the received message and the sender's digital signature. In digital signature schemes, the sender owns a secret key based on which the signature (which is public) is generated. An important aspect is that the sender cannot reject the message once the message is digitally signed by the sender. This property is called *nonrepudiation*, and it is crucial in digital signature schemes. Besides nonrepudiation, other important properties are *data integrity*, which ensures that the message has not been altered in its transit from source to destination, and *message authentication*, which allows the destinations of a message to confirm the identity of the source.

Although they belong to public-key cryptography, the signature schemes have a slightly different form than regular encryption schemes. A digital signature scheme has the following algorithms:

- **Key generation.** This algorithm generates a pair of keys, namely, the secret key and the public key.
- **Signing.** This algorithm is used to generate the digital signature based on the secret key and the message that needs to be signed.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_14

- **Verifying.** This algorithm is used to verify the authenticity of the message, based on the message itself, the public key, and the signature generated previously. The result of this algorithm is "accept" or "reject."

Digital signatures were first mentioned by Diffie and Hellman in [1], but their rigorous description and security requirements were presented by Goldwasser, Micali, and Rivest in [2]. The following are important milestones related to digital signatures [3, 4]:

- **1985.** The United States' National Institute of Standards and Technology (NIST) starts developing the Digital Signature Algorithm (DSA) as part of the Digital Signature Standard (DSS). DSA was later published in 1991 and became a Federal Information Processing Standard in 1993, providing a government-approved method for digital signatures.
- **1988.** Shafi Goldwasser, Silvio Micali, and Ronald Rivest publish a paper providing a rigorous definition and security requirements for digital signature schemes. The work [2] is pivotal in formalizing the security properties that digital signature schemes should possess, including unforgeability under chosen-message attacks.
- **2000s.** Digital signatures based on Elliptic Curve Cryptography, such as the Elliptic Curve Digital Signature Algorithm (ECDSA), gain popularity. ECC offers the same level of security as RSA but with shorter key lengths, making it more efficient and suitable for devices with limited computing resources.
- **2010s.** With the increasing threat of quantum computing, research intensifies on post-quantum cryptography, aiming to develop digital signature algorithms that can resist quantum attacks. The NIST starts the process of standardizing post-quantum cryptographic algorithms, including digital signature schemes.

- **2020s.** Digital signatures become crucial in the blockchain and smart contracts ecosystem, providing a secure and verifiable way to execute agreements and transactions without intermediaries. Technologies like Bitcoin and Ethereum demonstrate the utility of digital signatures in decentralized systems.

Implementations of Digital Signatures

Digital signatures can be stand-alone algorithms or a combination of an encryption algorithm with hash functions. Beware of the second approach as using only certified algorithms for this purpose is recommended. Our first example is using the RSA algorithm (learn about the RSA algorithm in Chapter 11) with the hash function SHA-256 (learn more about the hash function in Chapter 9). Listing 14-1 shows the implementation, and Figure 14-1 shows the output. Java implements a class called `Signature` from the `java.security` package that contains a wide range of digital signatures, for example, RSA combined with a hash function, NIST's Digital Signature Algorithm (DSA) [5] combined with a hash function, and the Edwards-Curve signature algorithm [6]. You can learn more about signature schemes implemented in Java in [7].

Listing 14-1. Example of Using Digital Signatures

```
1   package dsc;
2
3   import java.security.*;
4   import java.util.HexFormat;
5
6   public class digital_signature
7   {
8       //the function will help to generate the digital signature based on
        SHA256 and RSA algorithm, using an input and a private key
9       public static byte[] GenerateDS(byte[] dataInput, PrivateKey
        prvKey) throws Exception
10      {
11          var signSHA256RSA = Signature.getInstance("SHA256withRSA");
12          signSHA256RSA.initSign(prvKey);
```

```
13          signSHA256RSA.update(dataInput);
14          return signSHA256RSA.sign();
15      }
16
17      //the function will generate a asymmetric key pair based on
        SecureRandom class and using RSA algorithm
18      //128 value
19      public static KeyPair AKPGenRSA(int size) throws Exception
20      {
21          var sr = new SecureRandom();
22          var kpg = KeyPairGenerator.getInstance("RSA");
23          kpg.initialize(size, sr);
24          return kpg.generateKeyPair();
25      }
26
27      //function for verifying the digital signature based on a
        public key
28      public static boolean CheckDS(byte[] message, byte[]
        verifyingSignature, PublicKey pubKey) throws Exception
29      {
30          var signature = Signature.getInstance("SHA256withRSA");
31          signature.initVerify(pubKey);
32          signature.update(message);
33          return signature.verify(verifyingSignature);
34      }
35
36       //main function
37       public static void main(String args[]) throws Exception
38       {
39           var message = "Welcome To Apress. Enjoy learning
             cryptography.";
40           var kp512 = AKPGenRSA(512);
41           var kp1024 = AKPGenRSA(1024);
42           var kp2048 = AKPGenRSA(2048);
43
```

```
44          var generatedSignature512 = GenerateDS(message.getBytes(),
            kp512.getPrivate());
45          var generatedSignature1024 = GenerateDS(message.getBytes(),
            kp1024.getPrivate());
46          var generatedSignature2048 = GenerateDS(message.getBytes(),
            kp2048.getPrivate());
47
48          System.out.println("The message for which the signature will
            be computed is -> " + message);
49          System.out.println("The length of the message is -> " +
            message.length());
50
51          System.out.println("\n");
52
53          System.out.println("The 512-signature is -> \n " + bytesToHex(
            generatedSignature512));
54          System.out.println("Status of 512-verification -> " +
            CheckDS(message.getBytes(), generatedSignature512, kp512.
            getPublic()));
55
56          System.out.println("\n");
57
58          System.out.println("The 1024-signature is -> \n " + bytesToHex
            (generatedSignature1024));
59          System.out.println("Status of 1024-verification -> " +
            CheckDS(message.getBytes(), generatedSignature1024, kp1024.
            getPublic()));
60
61          System.out.println("\n");
62
63          System.out.println("The 2048-signature is -> \n " + bytesToHex
            (generatedSignature2048));
64          System.out.println("Status of 2048-verification -> " +
            CheckDS(message.getBytes(), generatedSignature2048, kp2048.
            getPublic()));
```

```
65        }
66
67        public static String bytesToHex(byte[] bytes) {
68                return HexFormat.of().formatHex(bytes);
69           }
70  }
```

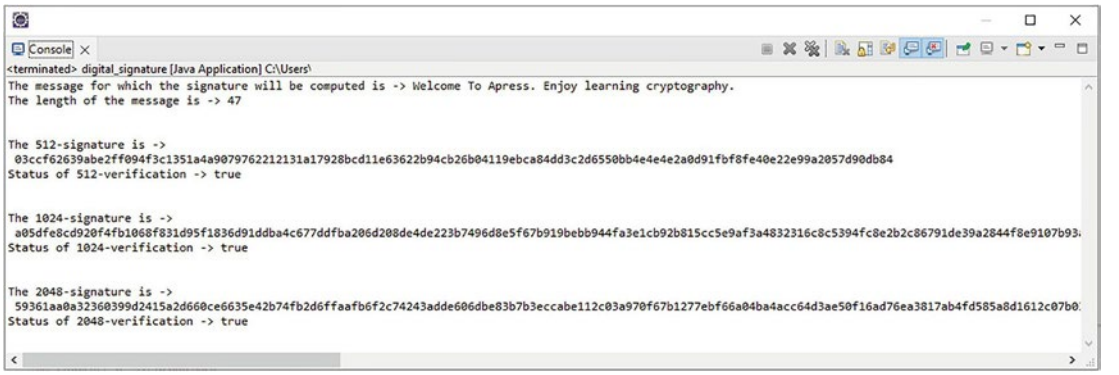

Console
<terminated> digital_signature [Java Application] C:\Users\
The message for which the signature will be computed is -> Welcome To Apress. Enjoy learning cryptography.
The length of the message is -> 47
The 512-signature is ->
03ccf62639abe2ff094f3c1351a4a90797622121 31a17928bcd11e63622b94cb26b04119ebca84dd3c2d6550bb4e4e4e2a0d91fbf8fe40e22e99a2057d90db84
Status of 512-verification -> true
The 1024-signature is ->
a05dfe8cd920f4fb1068f831d95f1836d91ddba4c677ddfba206d208de4de223b7496d8e5f67b919bebb944fa3e1cb92b815cc5e9af3a4832316c8c5394fc8e2b2c86791de39a2844f8e9107b93
Status of 1024-verification -> true
The 2048-signature is ->
59361aa0a32360399d2415a2d660ce6635e42b74fb2d6ffaafb6f2c74243adde606dbe83b7b3eccabe112c03a970f67b1277ebf66a04ba4acc64d3ae50f16ad76ea3817ab4fd585a8d1612c07b0
Status of 2048-verification -> true

Figure 14-1. *The result of Listing 14-1*

In lines 9–15, the digital signature is created. To do this, the message (`dataInput`) and the private key (`prvKey`) used to sign the message are both required. Here, an instance of the `Signature` class is created (in the previous example, the signature's type is `SHA256withRSA`), and then the signing process is initialized with the secret key. Next, the message is signed and then returned. In lines 22–25, the pair of the secret key and the public key for the RSA algorithm is generated. Next, in lines 28–34, the signature is verified, based on the message and the public key, following a process similar to the one for signing. In the main function, there are three examples of signing, namely, using RSA with the key length 512, then 1024, and finally 2048. In lines 67–69, there is a helper function that converts a string to a hex value, and a byte value to a hex value, respectively. In Listing 14-1, the use of `var` for local variables enhances readability and reduces verbosity without losing type safety. Also, `java.util.HexFormat` is used for byte-to-hex conversion. This API introduced is more efficient and readable compared to manual conversion methods.

The next signature scheme is ElGamal [8], which is based on the discrete logarithm problem. The protocol is described next.

Key generation.

- In this algorithm, the sender picks a large prime integer value p and a primitive root α.
- The sender picks an integer value a, which is the secret key, and then computes the value $\beta \equiv \alpha a \ (mod\ p)$.
- The public key is $pk = (p, \alpha, \beta)$, and the secret key is $sk = a$.

Signing the message.

- The sender picks randomly an integer value k for which $\gcd(k, p-1) = 1$ and then computes the value $y_1 \equiv \alpha^k (mod\ p)$ and the value $y_2 \equiv (m - ay_1)k^{-1} (mod\ (p-1))$, where m is the message that needs to be signed.

Verifying the message.

- To verify the message, the receiver verifies if the result of the following "equality" is true: $y_1^{y2} \bullet \beta^{y_1} \equiv \alpha^m \ (mod\ \ p)$.

In Listing 14-2 there is a naïve implementation of the ElGamal signature scheme; Figure 14-2 shows the result.

Listing 14-2. ElGamal Signature Scheme

```
1   import java.math.BigInteger;
2   import java.util.Scanner;
3
4   public class ElGamalSignature {
5       public static void main(String[] args) {
6           try (Scanner sc = new Scanner(System.in)) {
7               System.out.println("Value of p");
8               var p = new BigInteger(sc.next());
9               System.out.println("Value of alpha");
10              var alpha = new BigInteger(sc.next());
11              System.out.println("Value of m");
12              var m = new BigInteger(sc.next());
13              System.out.println("Value of k");
14              var k = new BigInteger(sc.next());
```

```
15
16              var sign = new signingAlgorithm(p, alpha, k, m);
17
18              System.out.println("Public key: " + p.intValue() + " " +
                alpha.intValue() + " " + sign.beta.intValue());
19              System.out.println("y1: " + sign.y1 + "; y2: " + sign.y2);
20
21              var verify = new verifyingAlgorithm(m, sign.y1, sign.y2,
                sign.alpha, sign.beta, p);
22              verify.verify();
23          }
24      }
25  }
26
27  class signingAlgorithm {
28      public BigInteger p, alpha, beta, k, m, y1, y2;
29      private BigInteger a = new BigInteger("3");
30
31      public signingAlgorithm(BigInteger p, BigInteger alpha,
        BigInteger k, BigInteger m) {
32          this.p = p;
33          this.alpha = alpha;
34          this.k = k;
35          this.m = m;
36
37          computeBeta();
38          computeY1();
39          computeY2();
40      }
41
42      public void computeBeta() {
43          this.beta = alpha.modPow(a, p);
44      }
45
46      public void computeY1() {
47          this.y1 = alpha.modPow(k, p);
```

```
48      }
49
50      public void computeY2() {
51          var invK = k.modInverse(p.subtract(BigInteger.ONE));
52          this.y2 = ((m.subtract(a.multiply(y1))).multiply(invK)).
            mod(p.subtract(BigInteger.ONE));
53      }
54  }
55
56  class verifyingAlgorithm {
57
58      public BigInteger m, y1, y2, alpha, beta, p;
59
60      public verifyingAlgorithm(BigInteger m, BigInteger y1,
        BigInteger y2, BigInteger alpha, BigInteger beta, BigInteger p) {
61          this.m = m;
62          this.y1 = y1;
63          this.y2 = y2;
64          this.alpha = alpha;
65          this.beta = beta;
66          this.p = p;
67      }
68
69      public void verify() {
70          var left = y1.modPow(y2, p).multiply(beta.modPow(y1, p)).
            mod(p);
71          var right = alpha.modPow(m, p);
72
73          if (left.equals(right))
74              System.out.println("Signature verified");
75          else
76              System.out.println("Signature mismatch");
77      }
78  }
```

Figure 14-2. *The result of signing a message with ElGamal*

Note that the implementation from Listing 14-2 just follows the steps described for the ElGamal signature scheme. As well as previously, the use of `var` makes the code more concise and readable. This feature doesn't change the logic or performance but improves code maintainability. For scanning, the input `Try-with-Resources` for `Scanner` was used to ensure the `Scanner` object is properly closed, enhancing resource management and preventing potential resource leaks. Although Listing 14-2 just follows the steps described for the ElGamal signature scheme, however, in practice, it is recommended to use algorithms already implemented in libraries, as they are tested and validated widely. In advanced implementations, the message is hashed using a hash function, and it's working with the hash value of the message instead of the plain message. You can learn more about signature schemes in [9] and [10].

Standards and Regulations of Digital Signatures

Digital signatures are recognized by law in many jurisdictions around the world, providing a legal framework that acknowledges their validity and enforceability in electronic transactions and documents. The legal recognition of digital signatures facilitates their use in a wide range of applications, from contracts to government documents. For example, the following regulations and standards are used in the United States:

- **Electronic Signatures in Global and National Commerce Act (ESIGN, 2000) [11].** This federal law provides that electronic signatures and records are legally valid and enforceable, equating them with their handwritten counterparts. The ESIGN Act ensures that a contract or signature cannot be denied legal effect solely because it is in electronic form.

- **Uniform Electronic Transactions Act (UETA, 1999) [12].** Adopted by the majority of U.S. states, UETA offers a legal framework for electronic signatures and records. It establishes that electronic signatures and records carry the same weight and legal effect as traditional paper documents and handwritten signatures, provided that all parties to a transaction agree to proceed electronically.

The following regulations and standards are used the European Union:

- **Electronic Identification and Trust Services for Electronic Transactions in the Internal Market (eIDAS, 2014) [13].** eIDAS provides a standardized framework across all EU member states for electronic identification and trust services, including digital signatures. The regulation distinguishes between electronic signatures, advanced electronic signatures (AdES), and qualified electronic signatures (QES), with the latter offering the highest security level and being recognized as equivalent to handwritten signatures across the EU. eIDAS also outlines the requirements for trust service providers, ensuring a high level of security and interoperability for electronic transactions within the EU.

Conclusion

This chapter presented digital signatures, which are important algorithms used to validate the bond between the sender and the message that it signs. The chapter included two implementations of digital signatures: the first is a combination of an encryption scheme (RSA) and a hash function (SHA256), and the second one is the implementation of the ElGamal signature scheme. Also, the chapter highlighted important standards and regulations that involve digital signatures.

On the other hand, digital signatures have great potential in blockchain technology. Smart contracts, which are self-executing contracts with the terms of the agreement directly written into lines of code, use digital signatures for initiating and validating transactions on the blockchain. These contracts automatically enforce and execute the terms of agreements based on the digital signatures provided, offering a new level of automation and efficiency in contractual obligations.

References

[1] Diffie, W., & Hellman, M. (1976). New directions in cryptography. IEEE transactions on Information Theory, 22(6), 644–654.

[2] Goldwasser, S., Micali, S., & Rivest, R. L. (1988). A digital signature scheme secure against adaptive chosen-message attacks. SIAM Journal on computing, 17(2), 281–308.

[3] M. N. S. Perera, T. Nakamura, M. Hashimoto, H. Yokoyama, C.-M. Cheng, and K. Sakurai, "A Survey on Group Signatures and Ring Signatures: Traceability vs. Anonymity," Cryptography, vol. 6, no. 1, Art. no. 1, Mar. 2022, doi: 10.3390/cryptography6010003.

[4] O. Manz, "Digital Signature," in Encrypt, Sign, Attack: A compact introduction to cryptography, O. Manz, Ed., Berlin, Heidelberg: Springer, 2022, pp. 87–123. doi: 10.1007/978-3-662-66015-7_4.

[5] FIPS PUB 186-4. Digital Signature Standard. Available online: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[6] Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., & Yang, B. Y. (2012). High-speed high-security signatures. Journal of cryptographic engineering, 2(2), 77–89.

[7] Signature Algorithms, https://docs.oracle.com/en/java/javase/17/docs/specs/security/standard-names.html#signature-algorithms.

[8] ElGamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE transactions on information theory, 31(4), 469–472.

[9] Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A. (2018). Handbook of applied cryptography. CRC Press.

[10] Stinson, D. R. (2005). Cryptography: theory and practice. Chapman and Hall/CRC.

[11] 'Electronic Signatures in Global and National Commerce Act (E-Sign Act) | NCUA'. [Online]. Available: https://ncua.gov/regulation-supervision/manuals-guides/federal-consumer-financial-protection-guide/compliance-management/deposit-regulations/electronic-signatures-global-and-national-commerce-act-e-sign-act.

[12] "Uniform Electronic Transaction Act." [Online]. Available: https://www.cga.ct.gov/2000/rpt/2000-R-1076.htm.

[13] "Regulation - 910/2014 - EN - e-IDAS - EUR-Lex." [Online]. Available: https://eur-lex.europa.eu/eli/reg/2014/910/oj.

CHAPTER 15

Lattice-Based Cryptography and NTRU

This chapter presents an important topic in cryptography, namely, the encryption schemes based on lattices. These structures are important because they have been proven they can be included in post-quantum cryptography. Lattices are one of the few techniques that can resist quantum attacks and are good candidates for encryption schemes for quantum computers.

Such techniques are important because quantum computers are knocking on the door, and the digital world needs to be prepared. Traditional encryption systems, such as RSA, Diffie-Hellmann, and elliptic curves cryptosystems, will be overcome by the technological revolution brought about by quantum computers; therefore, more powerful techniques and tools are necessary to protect the data.

As great as lattices seem, we need to pay a price for using lattices: they are very complex mathematical structures that require solid knowledge of algebra and the ability to understand and work with abstract concepts. Lattices are the basis for a large number of fully homomorphic encryption schemes, which are another hot topic in cryptography nowadays, because they allow computations to be made directly on the encrypted data, obtaining the same result as applying equivalent operations on the plaintext (see Chapter 14 for more details). Examples of lattice-based cryptosystems are Peikert's RLWE Key-Exchange [1], the GGH cryptosystem [2], and the NTRU cryptosystem [3].

This chapter will cover the main mathematical aspects that characterize lattices. The concepts presented represent the minimum theoretical information required to understand lattices.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_15

Consider the n-dimensional real numbers space $\mathbb{R}^n$, whose elements have the form of a row vector $v = (v_1, \ldots, v_n)$ and each $v_i \in \mathbb{R}$, $i \in \{1, \ldots, n\}$. A lattice L has the following representation:

$$\mathcal{L}(v) = \left\{ \sum_{i=1}^{n} a_i v_i | a_i \in \mathbb{Z} \right\}.$$

Note that the vector v from the previous definition should be a basis in R^n, and a_i is an integer number. In other words, a lattice is the set of all linear combinations with integer coefficients. From the lattice's definition, it can be easily seen that Z^n is a lattice produced by the standard basis in R^n. Figure 15-1 shows the representation of a lattice in R^2.

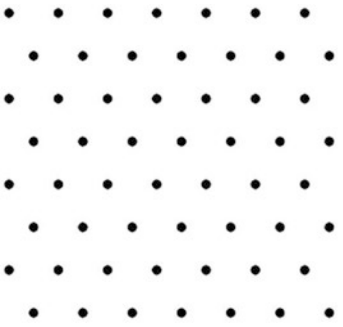


Figure 15-1. *Example of a lattice (source: https://en.wikipedia.org/wiki/Lattice_(group)#/media/File:Equilateral_Triangle_Lattice.svg)*

Also of interest to cryptography are the computational problems regarding lattices, which lead to the hardness assumptions of cryptosystems. Several computing problems involve lattices: shortest vector problem (SVP), short basis problem (SBP), short independent vector problem (SIVP), or closest vector problem (CVP).

The SVP is as follows: let V be a vector space, b a basis in V, and N a norm. Given a lattice L(b), a vector $v \in V$ should be found for which its norm in V is the minimum distance in L. Alternatively, v should have the following form:

$$\|v\| = \lambda(\mathcal{L}(b)),$$

where ||. || denotes the norm in V and λ is the minimum distance defined in L(b).

From here, two types of SVP result:

- **Calculational SVP.** Finding the minimum distance $\lambda(\mathrm{L}(b))$, when b and $\mathrm{L}(b)$ are given
- **Decisional SVP.** Deciding whether $\lambda(\mathrm{L}(b)) \leq d$ or $\lambda(\mathrm{L}(b)) > d$, when b, $\mathrm{L}(b)$, and $d \in \mathrm{R}$, $d > 0$ are given

This may seem easy to solve, but when it is working with a large dimension basis, the problem is complicated. For example, in cryptography, the dimensions of the lattices can reach 10,000 instead of just 2 as in the previous example. Therefore, each point will have 10,000 components, and for each component, the conditions from earlier must be fulfilled *simultaneously*.

To test whether an encryption system is quantum-resistant, it must be resistant to attacks based on Shor's algorithm [4].

Among the first cryptosystems that used lattices was the NTRU encryption system [3], which is based on SVP. More exactly, the SVP for NTRU lies in factoring specific polynomials in two polynomials with extremely low coefficients. Those specific polynomials come from a truncated polynomial ring. The algorithms for the NTRU cryptosystem are as follows [3]:

- **Key generation.** Consider $n \in \mathrm{N}$, f, g polynomials with a maximum degree $n - 1$ and with coefficients in the set $\{-1, 0, 1\}$. In addition, for f, two polynomials f_p and f_q should exist such that $f \circ f_p = 1(mod\ p)$ and $f \circ f_q = 1(mod\ q)$. The public key is $PK = p \circ f_q \circ g\ (mod\ q)$, and the private key is $SK = (f, f_p, g)$.
- **Encryption.** The message has the form of a polynomial m with coefficients in $\{-1, 0, 1\}$. Generate randomly a polynomial r with low coefficients, keep it private, and encrypt the message as $c = r \circ PK + m\ (mod\ q)$.
- **Decryption.** For decryption, the message is recovered using the computations shown here:

$$a = f^{\circ}c\left(mod\ \ q\right)$$

$$b = a\left(mod\ \ p\right)$$

$$m = f_p{}^{\circ}b$$

Practical Implementation of the NTRU Library

The following example of NTRU is based on [5]. The example has been adjusted and commented accordingly to show how the NTRU library can be used in real scenarios by providing encryption and decryption operations over messages. The current library from [5] is used as an extension in the Bouncy Castle library as well. The implementation complies with FIPS regulations for Java 1.7 and 1.8 and Java 11 [6].

Listing 15-1 shows the full implementation and commented version of the encryption and decryption operations. In the example provided in Listing 15-1, we have used the net.sf.ntru library, which can be downloaded from `https://github.com/tbuktu/ntru`. To add this as a library to Eclipse, complete the following steps before running the example from Listing 15-1:

1. Right-click the project and navigate to Build Path ➤ Configure Build Path (see Figure 15-2).

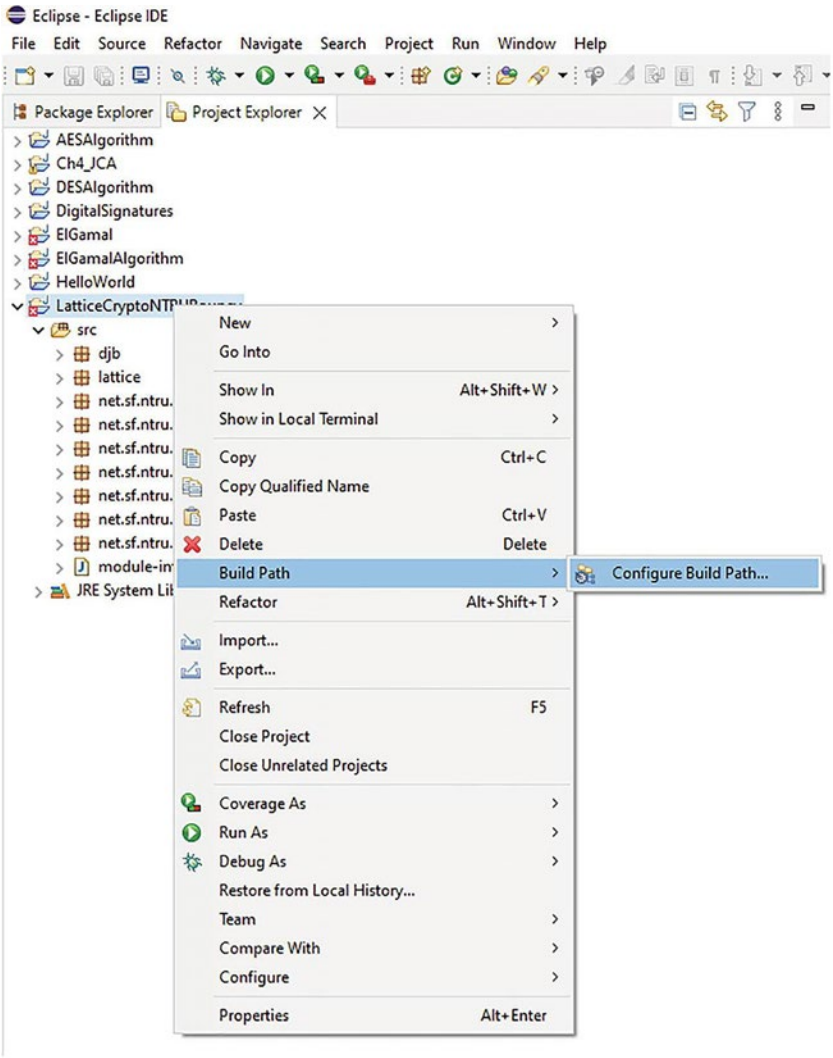


Figure 15-2. *Configuring the building path for adding NTRU library from* https://github.com/tbuktu/ntru

2. Click Add External Class Folder. Select the path to the location where the NTRU library was downloaded (Figure 15-3).

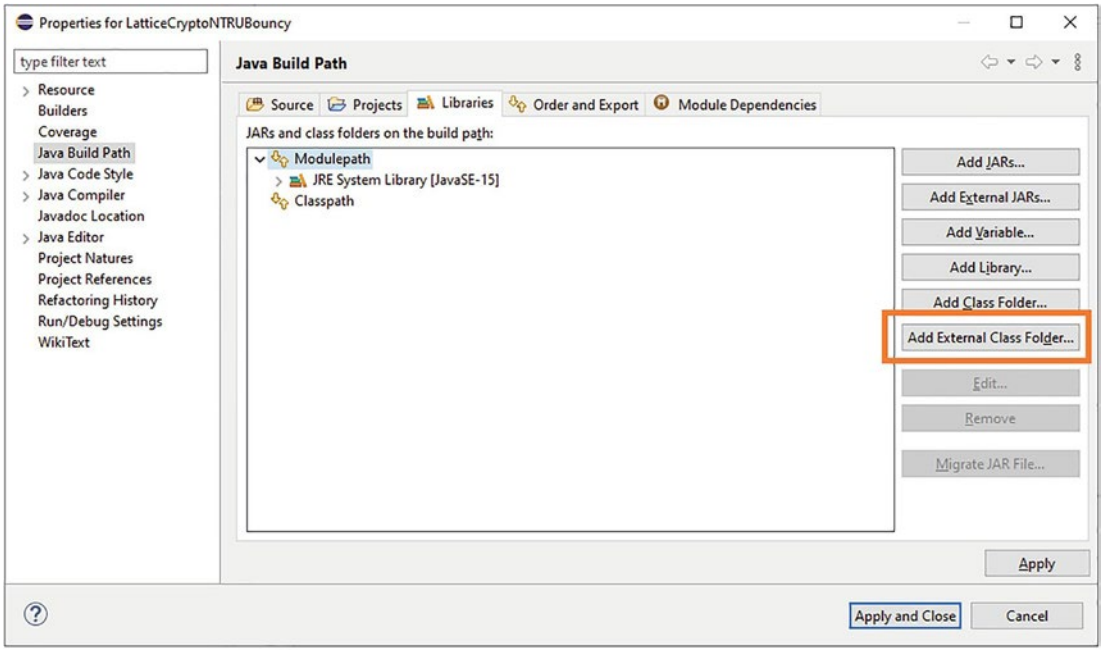


Figure 15-3. *Adding an external class folder for the NTRU library*

3. After adding it to the JARs and class folders on the build path list, check that the path to the NTRU has been added correctly. Once everything is OK, click Apply and Close (Figure 15-4).

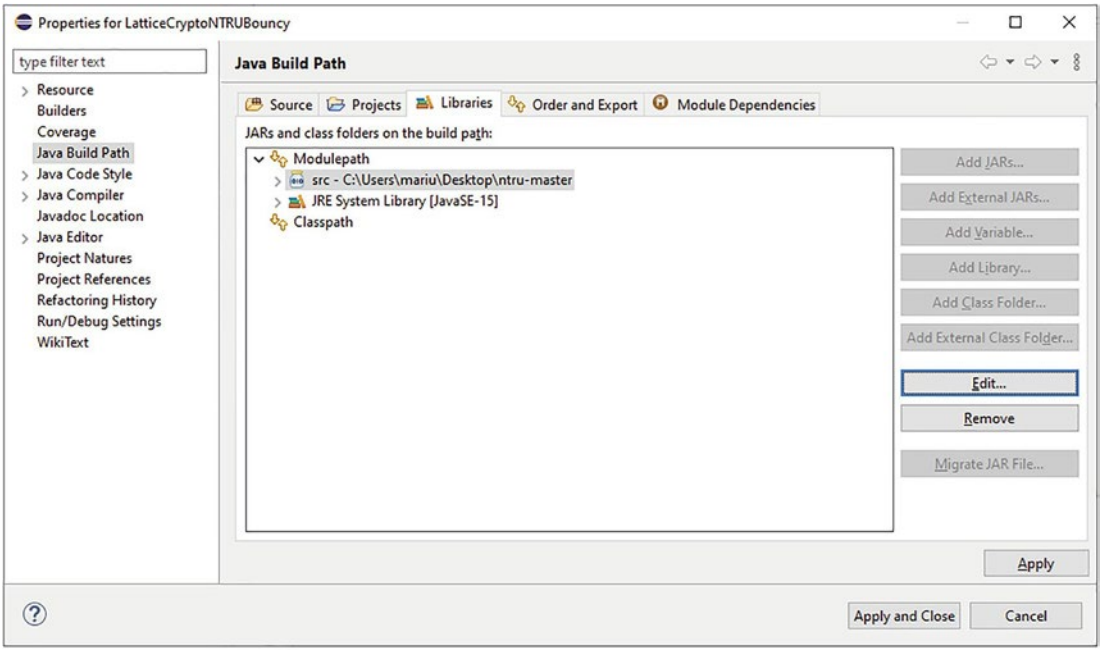


Figure 15-4. *Added NTRU library*

Once steps 1 through 3 have been performed, you should be able to run the example from Listing 15-1 smoothly without any issues.

Listing 15-1. Full Implementation of NTRU Operations, Encryption, and Decryption

```
import java.util.Arrays;
import javax.crypto.*;
import javax.crypto.spec.*;
import net.sf.ntru.encrypt.*;

public class AesNtru
{

    public static void main(String[] args) throws Exception
    {
        new ExampleOf_AES_NTRU().ExecuteNTRUEncryptionDecryption
        Process();
    }
```

```
13
14      private void ExecuteNTRUEncryptionDecryptionProcess() throws
        Exception
15      {
16          String message = "Welcome to Apress. Enjoy learning " + "
            practical cryptography and NTRU operations";
17
18          // setup the parameters for NTRU and AES
19          String ntru_aes_parameters = "AES/CBC/PKCS5Padding";
20          int length_for_aes = 128;
21          EncryptionParameters parameters_for_ntru =
            EncryptionParameters.APR2011_439_FAST;
22
23          // setup an NTRU instance based on the encryption parameters
24          // and generate the key pair using NTRU instance
25          NtruEncrypt encryption_with_ntru = new
            NtruEncrypt(parameters_for_ntru);
26          EncryptionKeyPair ntru_key_pair = encryption_with_ntru.
            generateKeyPair();
27
28
29          System.out.println("Decrypted message = " + message.
            substring(0, 50) + "...");
30          System.out.println("Length of plain message = " + message.
            length());
31          System.out.println("Maximum length of NTRU = " + parameters_
            for_ntru.getMaxMessageLength());
32
33          // compute the encryption of the message
34          byte[] encryption_message = EncryptTheMessage(message.
            getBytes(),
35                    ntru_key_pair.getPublic(),
36                    ntru_aes_parameters,
37                    length_for_aes,
38                    parameters_for_ntru);
```

```
39
40          System.out.println("Encrypted length = " + encryption_
            message.length +
41                  " (NTRU=" + parameters_for_ntru.
                    getOutputLength() + ", "
42                  + "AES=" + (encryption_message.length - parameters_
                    for_ntru.getOutputLength()) + ")");
43
44          // compute the decryption of the message
45          String decryption_message = new String(DecryptTheMessage
            (encryption_message,
46                    ntru_key_pair,
47                    ntru_aes_parameters,
48                    length_for_aes,
49                    parameters_for_ntru));
50
51          System.out.println("The decryption of message is   = " +
            decryption_message.substring(0, 50) + "...");
52          System.out.println("The length of the decrypted message is =
            " + decryption_message.length());
53      }
54
55      // encryption function will receive the following parameters:
56      // - the public key
57      // - the mode of AES
58      // - the length of AES
59      // - the encryption parameters for NTRU block
60      private byte[] EncryptTheMessage(byte[] clearMessage,
61                EncryptionPublicKey public_key,
62                String modeOfAES,
63                int lengthOfAES,
64                EncryptionParameters ntru_parameters) throws Exception
65      {
66           // compute cryptographic AES key
67          SecretKey cryptoKeyForAES = generateAesKey(lengthOfAES);
68
```

```
69          // generate key specifications for encoding with AES - also
               it will for generating
70          // the initialization vector (IV)
71          SecretKeySpec key_specifications_aes = new SecretKeySpec
            (cryptoKeyForAES.getEncoded(), "AES");
72
73          // providing encryption for message using AES
74          Cipher algorithm = Cipher.getInstance(modeOfAES);
75          algorithm.init(Cipher.ENCRYPT_MODE, key_specifications_aes);
76          byte[] initialization_vector = algorithm.getParameters().getP
            arameterSpec(IvParameterSpec.class).getIV();
77          byte[] encryption_with_aes = algorithm.doFinal(clearMessage);
78
79          // encrypt AES key and IV with NTRU
80          NtruEncrypt encryption_with_ntru = new NtruEncrypt(ntru_
            parameters);
81          byte[] cryptoAESKey_Array = cryptoKeyForAES.getEncoded();
82          byte[] initializationVector_and_cryptoKey = generate_byte_
            array(cryptoAESKey_Array, initialization_vector);
83          byte[] encryptedResultWithNtru = encryption_with_ntru.
            encrypt(initializationVector_and_cryptoKey, public_key);
84
85          // put everything in one byte array
86          return generate_byte_array(encryptedResultWithNtru,
            encryption_with_aes);
87      }
88
89
90      // decryption function will receive the following parameters
91      // - the encrypted message as an array of byte
92      // - the decryption key pair
93      // - the mode of AES
94      // - the length of AES
95      // - the encryption parameters related to NTRU block
96      private byte[] DecryptTheMessage(byte[] encrypted_message,
```

```
97                EncryptionKeyPair key_pair_for_encryption,
98                String modeOfAES,
99                int lengthOfAES,
100               EncryptionParameters ntru_parameters) throws Exception
101       {
102            // set the encrypted ntru block based on the NTRU parameters
103            NtruEncrypt encrypted_ntru_block = new NtruEncrypt(ntru_
               parameters);
104
105           // obtain crypto key and initialization vector by decrypting
                 the NTRU block
106           byte[] encrypted_block_with_ntru = Arrays.copyOf(encrypted_
              message, ntru_parameters.getOutputLength());
107           byte[] arrayOfKeyAndIV = encrypted_ntru_block.
              decrypt(encrypted_block_with_ntru, key_pair_for_encryption);
108           byte[] arrayOfCryptoAESKey = Arrays.copyOf(arrayOfKeyAndIV,
              lengthOfAES/8);
109           byte[] initializationVectorArray = Arrays.
              copyOfRange(arrayOfKeyAndIV, lengthOfAES/8, 2*lengthOfAES/8);
110
111           // based on the AES crypto key and initialization vector,
              perform the decryption of the message
112           byte[] encrypted_message_with_aes = Arrays.
              copyOfRange(encrypted_message,
113                       encrypted_block_with_ntru.length,
114                       encrypted_message.length);
115
116           // specify the encryption mode of AES
117           Cipher algorithm = Cipher.getInstance(modeOfAES);
118
119           // configure the key specification related to the algorithm
                 that we are using (e.g., AES)
120           SecretKeySpec key_specification_aes = new SecretKeySpec
              (arrayOfCryptoAESKey, "AES");
121
```

```
122             // set the specifications of the parameters for the
                   initialization vector
123             IvParameterSpec initialization_vector_specifications = new Iv
                ParameterSpec(initializationVectorArray);
124
125
126             // initialize the algorithm for decryption by specifying the
                   mode, the AES key and initialization vector
127             algorithm.init(Cipher.DECRYPT_MODE, key_specification_aes,
                initialization_vector_specifications);
128
129             // obtain the message in clear based on the encrypted message
                   with AES
130             byte[] messageInClear = algorithm.doFinal(encrypted_message_
                with_aes);
131
132             // return the clear version of the message
133             return messageInClear;
134         }
135
136         private SecretKey generateAesKey(int number_of_bits) throws
                Exception
137         {
138              // generate the key for AES
139             KeyGenerator generatorForAESKey = KeyGenerator.
                getInstance("AES");
140
141             // initialize the key based on the number of the bits
142             generatorForAESKey.init(number_of_bits);
143
144             // get and return the generated crypto key
145             return generatorForAESKey.generateKey();
146         }
147
```

```
148        // based on two arrays (byteArray1 and byteArray2) generate a
              third one (byteArray3) by concatenate
149        // the encrypted result with NTRU and encrypted result with AES
150        private byte[] generate_byte_array(byte[] byteArray1, byte[]
           byteArray2)
151        {
152            // the final result of concatenation of byteArray1 (encrypted
                  result with NTRU)
153             // and byteArray2 (encrypted result with AES)
154             byte[] byteArray3 = new byte[byteArray1.length + byteArray2.
                length];
155
156             // perform the concatenation
157            System.arraycopy(byteArray1, 0, byteArray3, 0, byteArray1.
               length);
158            System.arraycopy(byteArray2, 0, byteArray3, byteArray1.
               length, byteArray2.length);
159
160            // return the third-byte array containing the result of the
                  concatenation
161            return byteArray3;
162              }
163         }
```

When running the example from Listing 15-1, the output should look like Figure 15-5.

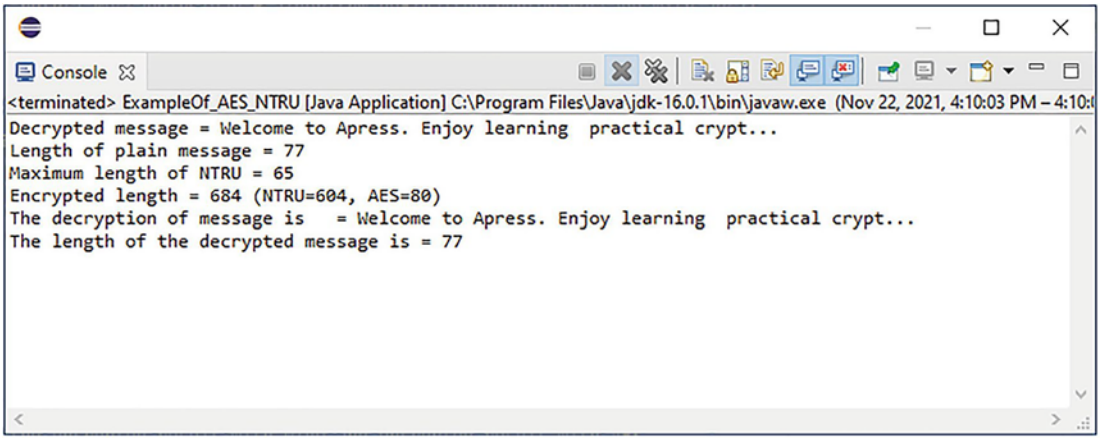

Console
<terminated> ExampleOf_AES_NTRU [Java Application] C:\Program Files\Java\jdk-16.0.1\bin\javaw.exe (Nov 22, 2021, 4:10:03 PM – 4:10:
Decrypted message = Welcome to Apress. Enjoy learning practical crypt...
Length of plain message = 77
Maximum length of NTRU = 65
Encrypted length = 684 (NTRU=604, AES=80)
The decryption of message is = Welcome to Apress. Enjoy learning practical crypt...
The length of the decrypted message is = 77

Figure 15-5. *The output of the NTRU encryption and decryption process*

Except for the main function, which executes the NTRU encryption and decryption mechanisms, the process is based on these five functions:

```
private void ExecuteNTRUEncryptionDecryptionProcess()
private byte[] EncryptTheMessage()
private byte[] DecryptTheMessage()
private SecretKey generateAesKey()
private byte[] generate_byte_array()
```

In the following paragraphs, we will examine only three functions based on their parameters and purpose. The other two functions are quite self-explanatory. The scenario is pretty simple and straightforward; the main point for the NTRU implementation is represented by the `ExecuteNTRUEncryptionDecryptionProcess()` function, as shown in Listing 15-2.

The first step is represented in line 3 where we are declaring a variable that will hold the message that we want to encrypt. To achieve the encryption process, we will need to set the parameters for NTRU and AES as mentioned in line 6. In line 7, we will set the length of the encryption key to 128 bits, and in line 8 we will mention through `EncryptionParameters` the parameter `APR2011_439_FAST`. Once the parameter is set, it will give 128 bits of security. The mode is designed to work using product-form polynomials [7]. Moving forward, in lines 10 and 11, we set up an NTRU instance using the encryption parameters and cryptographic key generated earlier using the NTRU instance and its given parameters (line 8). Next, in lines 14 to 16 we will show different aspects related to the process (as shown in Figure 15-1, the first three lines

from the console). In line 18 we declare a byte array for the encrypted message, and the encryption of the message is obtained using the `EncryptTheMessage()` method (described and shown in Listing 15-3). This process has to be done with respect to the parameters, such as the message (its representation as bytes), the public key from the key pair generated in line 11, the NTRU AES parameters set in line 6, the length of an encryption key for AES (set in line 7), and the parameters mentioned for NTRU (set in line 8). In line 24 we will show in the console the result (as shown in Figure 15-1, the fourth line from the console output). Continue with line 29, we will process the decryption process for the encrypted message with the help of the `DecryptTheMessage()` method, shown and described in Listing 15-4. The parameters received by the `DecryptTheMessage()` method are the same as the ones from the `EncryptTheMessage` method, except for the first parameter, which is represented by the encryption form of the message. In lines 35 and 37, we will show the result, as displayed in Figure 15-1, which are the last two lines from the output.

Listing 15-2. ExecuteNTRUEncryptionDecryptionProcess() Function

```
1   private void ExecuteNTRUEncryptionDecryptionProcess() throws Exception
2   {
3           String message = "Welcome to Apress. Enjoy learning " + "
            practical cryptography and NTRU operations";
4
5           String ntru_aes_parameters = "AES/CBC/PKCS5Padding";
6           int length_for_aes = 128;
7           EncryptionParameters parameters_for_ntru =
            EncryptionParameters.APR2011_439_FAST;
8
9           NtruEncrypt encryption_with_ntru = new NtruEncrypt(parameters_
            for_ntru);
10          EncryptionKeyPair ntru_key_pair = encryption_with_ntru.
            generateKeyPair();
11
12          System.out.println("Decrypted message = " + message.
            substring(0, 50) + "...");
13          System.out.println("Length of plain message = " + message.
            length());
```

```
14          System.out.println("Maximum length of NTRU = " + parameters_
            for_ntru.getMaxMessageLength());
15
16          byte[] encryption_message = EncryptTheMessage(message.
            getBytes(),
17                    ntru_key_pair.getPublic(),
18                    ntru_aes_parameters,
19                    length_for_aes,
20                    parameters_for_ntru);
21
22          System.out.println("Encrypted length = " + encryption_message.
            length +
23                   " (NTRU=" + parameters_for_ntru.
                     getOutputLength() + ", "
24                   + "AES=" + (encryption_message.length - parameters_
                     for_ntru.getOutputLength()) + ")");
25
26          String decryption_message = new String(DecryptTheMessage
            (encryption_message,
27                    ntru_key_pair,
28                    ntru_aes_parameters,
29                    length_for_aes,
30                    parameters_for_ntru));
31
32          System.out.println("The decryption of message is   = " +
            decryption_message.substring(0, 50) + "...");
33          System.out.println("The length of the decrypted message is = "
            + decryption_message.length());
34      }
```

We will continue our explanation with the `EncryptTheMessage` method, which is shown in Listing 15-3 and explained further here. The method is private, and the returning value is represented by a byte array that represents the encrypted message.

In line 1 we have the signature of the method and its parameters/arguments, as follows:

- `clearMessage`: Represents the message in clear plaintext
- `public_key`: Represents the cryptographic public key
- `modeOfAES`: Represents the mode chosen for AES (set in line 6 from Listing 15-2)
- `lengthOfAES`: Represents the length of the cryptography key for AES in bits (set in line 7 from Listing 15-2)
- `ntru_parameters`: Represents the NTRU parameters (set in line 8 from Listing 15-2)

Based on these parameters, the processes in Listing 15-3 are executed by following a simple, logical, and rational procedure. In line 7 we will compute the AES cryptography key using the length received as a parameter. In line 9 we use the key specifications to help us to provide the proper encoding using AES, and we will obtain later the initialization vector, which is very important for AES. Further, in line 12 we mention an instance of class `Cipher` by providing the operating mode for AES. In line 14 we compute and obtain the initialization vector. Lines 18 to 23 represent the encryption process using the AES cryptography key and initialization vector with NTRU.

Listing 15-3. The EncryptTheMessage Method

```
1   private byte[] EncryptTheMessage(byte[] clearMessage,
2                 EncryptionPublicKey public_key,
3                 String modeOfAES,
4                 int lengthOfAES,
5                 EncryptionParameters ntru_parameters) throws Exception
6       {
7           SecretKey cryptoKeyForAES = generateAesKey(lengthOfAES);
8
9           SecretKeySpec key_specifications_aes = new
            SecretKeySpec(cryptoKeyForAES.getEncoded(), "AES");
10
11          Cipher algorithm = Cipher.getInstance(modeOfAES);
12          algorithm.init(Cipher.ENCRYPT_MODE, key_specifications_aes);
```

```
13          byte[] initialization_vector = algorithm.getParameters().
            getParameterSpec(IvParameterSpec.class).getIV();
14          byte[] encryption_with_aes = algorithm.doFinal(clearMessage);
15
16           NtruEncrypt encryption_with_ntru = new NtruEncrypt(ntru_
             parameters);
17          byte[] cryptoAESKey_Array = cryptoKeyForAES.getEncoded();
18          byte[] initializationVector_and_cryptoKey = generate_byte_
            array(cryptoAESKey_Array, initialization_vector);
19          byte[] encryptedResultWithNtru = encryption_with_ntru.
            encrypt(initializationVector_and_cryptoKey, public_key);
20
21          return generate_byte_array(encryptedResultWithNtru,
            encryption_with_aes);
22      }
```

Next, we will take a closer look at the `DecryptTheMessage` method, which is shown in Listing 15-4. The parameters received in line 1 through the signature of the method are similar to the ones from the `EncryptTheMessage` method shown in Listing 15-3.

Based on the NTRU parameters discussed, in line 7 we need those parameters to provide the proper decryption process. Lines 9 to 15 represent the obtaining process of the cryptographic key and initialization vector using the NTRU block. Moving forward, in line 18, based on the AES cryptographic key and initialization vector, we will perform the decryption of the message. In line 21 we will get an instance to set up the algorithm used in the decryption process and initialized in the variable `modeOfAES`. Further, in line 23, we will need to specify the key specifications that are related to the algorithm that we are using (e.g., AES); in line 25, we set the specification parameters for the initialization vector; and in line 28, we initialize the algorithm for decryption by specifying the mode, the cryptographic key, and the initialization vector. Moving forward, in line 31 we compute the decryption process and obtain the plaintext, and in line 33 we return the message.

Listing 15-4. DecryptTheMessage Method

```
1   private byte[] DecryptTheMessage(byte[] encrypted_message,
2                 EncryptionKeyPair key_pair_for_encryption,
3                 String modeOfAES,
```

```
4                int lengthOfAES,
5                 EncryptionParameters ntru_parameters) throws Exception
6        {
7            NtruEncrypt encrypted_ntru_block = new NtruEncrypt(ntru_
             parameters);
8
9            byte[] encrypted_block_with_ntru = Arrays.copyOf(encrypted_
             message, ntru_parameters.getOutputLength());
10           byte[] arrayOfKeyAndIV = encrypted_ntru_block.
             decrypt(encrypted_block_with_ntru, key_pair_for_encryption);
11           byte[] arrayOfCryptoAESKey = Arrays.copyOf(arrayOfKeyAndIV,
             lengthOfAES/8);
12           byte[] initializationVectorArray = Arrays.
             copyOfRange(arrayOfKeyAndIV, lengthOfAES/8, 2*lengthOfAES/8);
13
14           byte[] encrypted_message_with_aes = Arrays.
             copyOfRange(encrypted_message,
15                       encrypted_block_with_ntru.length,
16                       encrypted_message.length);
17
18           Cipher algorithm = Cipher.getInstance(modeOfAES);
19
20           SecretKeySpec key_specification_aes = new SecretKeySpec
             (arrayOfCryptoAESKey, "AES");
21
22           IvParameterSpec initialization_vector_specifications = new IvP
             arameterSpec(initializationVectorArray);
23
24           algorithm.init(Cipher.DECRYPT_MODE, key_specification_aes,
             initialization_vector_specifications);
25
26           byte[] messageInClear = algorithm.doFinal(encrypted_message_
             with_aes);
27
28           return messageInClear;
29       }
```

Secure Application Development

Implementing lattice-based encryption techniques in Java applications for different domains, such as online, enterprise, and mobile apps, requires a strong knowledge of cryptographic and application development principles. Lattice-based cryptography provides a secure layer for handling sensitive data operations that is resistant to quantum attacks, ensuring future-proof security.

For web applications that require secure data storage, lattice-based schemes can encrypt database entries on the server side before storage. The NTRU encryption library can be used to encrypt user-sensitive data, such as personal identifiers and financial information, before storing it in a database. Java's Cipher class can be adapted to use NTRU instead of traditional ciphers by integrating with a Java library that supports NTRU, as in the following code snippet:

```
public static byte[] encryptData(String data, PublicKey publicKey) throws
GeneralSecurityException {
  Cipher cipher = Cipher.getInstance("NTRU", new BouncyCastleProvider());
  cipher.init(Cipher.ENCRYPT_MODE, publicKey);
  return cipher.doFinal(data.getBytes(StandardCharsets.UTF_8));
}
```

In enterprise applications, lattice-based encryption can be used to exchange keys and manage session keys for secure communication between distributed services.

For example, a key exchange mechanism using lattice-based algorithms to securely exchange session keys between client and server or between microservices can be implemented as follows:

```
public static KeyPair generateNtruKeyPair() throws
GeneralSecurityException {
  KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("NTRU",
  new BouncyCastleProvider());
  return keyPairGenerator.generateKeyPair();
}
```

Lattice-based cryptography is typically more computationally expensive compared to conventional approaches. It is important to compare cryptographic methods to make sure they do not significantly decrease application performance, especially in mobile situations with limited resources. The following are some best practices:

- **Key Management.** Securely manage cryptographic keys using hardware security modules (HSMs) or secure enclave technology.
- **Regular Updates.** Keep cryptographic libraries up-to-date to mitigate vulnerabilities as they are discovered.
- **Audit and Compliance.** Regularly audit the cryptographic implementations and comply with relevant standards and regulations, especially in enterprise environments.

Developers can enhance application security by integrating lattice-based cryptography into Java apps, making them resistant to current and future threats.

Conclusion

This chapter presented lattices, one of the most important techniques in cryptography. Their importance in quantum cryptography was highlighted, as well as the reasons why lattices are one of the best candidates for post-quantum encryption schemes.

The technical presentation of lattices and the NTRU encryption scheme was followed by a practical example of using NTRU in Java.

References

[1] Peikert, C. (2014, October). Lattice cryptography for the internet. In International workshop on post-quantum cryptography (pp. 197–219). Springer, Cham.

[2] Goldreich, O., Goldwasser, S., & Halevi, S. (1997, August). Public-key cryptosystems from lattice reduction problems. In Annual International Cryptology Conference (pp. 112–131). Springer, Berlin, Heidelberg.

[3] Hoffstein, J., Pipher, J., & Silverman, J. H. (1998, June). NTRU: A ring-based public key cryptosystem. In International Algorithmic Number Theory Symposium (pp. 267–288). Springer, Berlin, Heidelberg.

[4] Shor, P. W. (1994, November). Algorithms for quantum computation: discrete logarithms and factoring. In Proceedings 35th annual symposium on foundations of computer science (pp. 124–134). IEEE.

[5] Java Implementation of NTRUEncrypt. GitHub repository: https://github.com/tbuktu/ntru.

[6] Bouncy Castle, FIPS regulations: https://bouncycastle.org/fips_faq.html.

[7] Class SignatureParameters. Available online: http://javadox.com/net.sf.ntru/ntru/1.2/net/sf/ntru/sign/SignatureParameters.html.

CHAPTER 16

Cryptography Tools

This chapter presents two of the most popular software applications and tools for cryptography that are great for learning and practice purposes: CrypTool (CT) [1] and OpenSSL [2].

CryptTool

The original version of CT was written in C++, but there are several versions, including an online version and a version written in Java, called JCrypTool (JCT). The latest stable version is 1.0.9, and it is supported on Windows, macOS, and Linux; it can be downloaded from the official website at `https://www.cryptool.org/en/jct/downloads`. After downloading the archive, unarchive it and then launch the executable file called `JCrypTool.exe`. After opening it, you'll see the main window of JCT, as shown in Figure 16-1. Here, there are several options. To start using JCT, click the Start JCT option.

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0_16

Figure 16-1. *The main window of JCT*

You'll see four main sections after starting to use JCT, as shown in Figure 16-2. The first section is File Explorer, which, by default, shows the content of the `Users` folder. The second section is the Help window, where you can find descriptions of different functionalities of JCT . The third section is a `.txt` file that can be written with different input values for a chosen cryptosystem. Lastly, the fourth section contains a list of encryption systems and cryptographic primitives that can be tested; it is called CryptoExplorer. You can choose different operations in CryptoExplorer such as Analysis, Visuals, or Games. For example, if you select an encryption system from the fourth section, then in the Help section you will see information about the selected topic in the window shown in Figure 16-3.

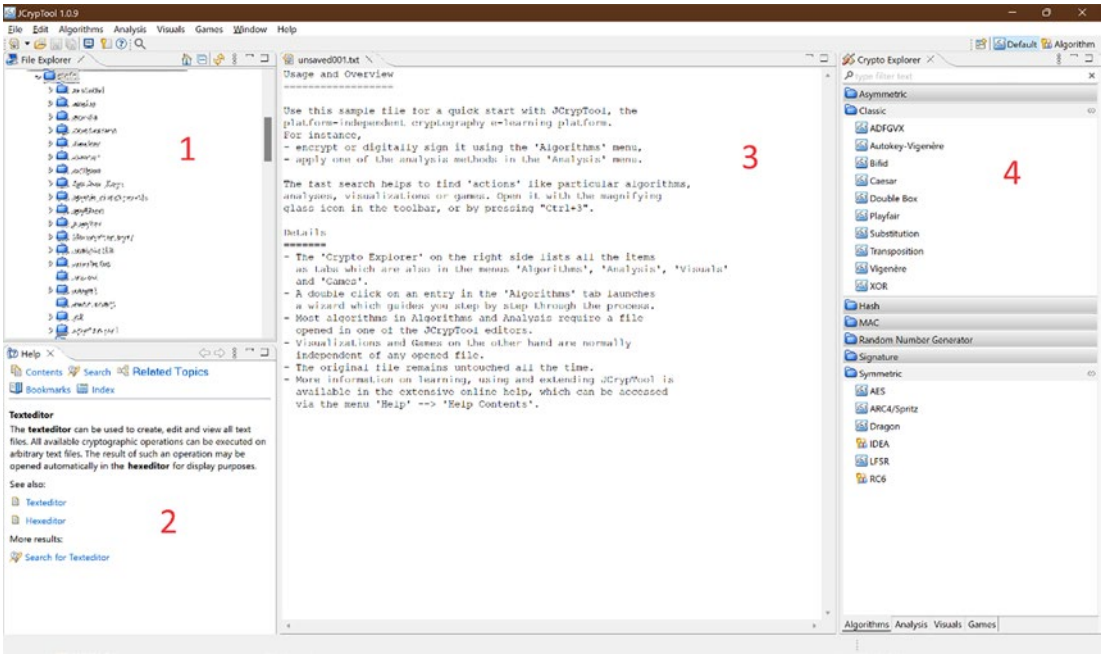

JCrypTool 1.0.9
File Edit Algorithms Analysis Visuals Games Window Help
File Explorer
1
unsaved001.txt
Usage and Overview
Use this sample file for a quick start with JCrypTool, the
platform-independent cryptography e-learning platform.
For instance,
- encrypt or digitally sign it using the 'Algorithms' menu,
- apply one of the analysis methods in the 'Analysis' menu.
The fast search helps to find 'actions' like particular algorithms,
analyses, visualizations or games. Open it with the magnifying
glass icon in the toolbar, or by pressing "Ctrl+3".
Details
- The 'Crypto Explorer' on the right side lists all the items
as tabs which are also in the menus 'Algorithms', 'Analysis', 'Visuals'
and 'Games'.
- A double click on an entry in the 'Algorithms' tab launches
a wizard which guides you step by step through the process.
- Most algorithms in Algorithms and Analysis require a file
opened in one of the JCrypTool editors.
- Visualizations and Games on the other hand are normally
independent of any opened file.
- The original file remains untouched all the time.
- More information on learning, using and extending JCrypTool is
available in the extensive online help, which can be accessed
via the menu 'Help' --> 'Help Contents'.
3
Help
Contents Search Related Topics
Bookmarks Index
Texteditor
The texteditor can be used to create, edit and view all text
files. All available cryptographic operations can be executed on
arbitrary text files. The result of such an operation may be
opened automatically in the hexeditor for display purposes.
See also:
Texteditor
Hexeditor
More results:
Search for Texteditor
2
Crypto Explorer
4
Asymmetric
Classic
ADFGVX
Autokey-Vigenère
Bifid
Caesar
Double Box
Playfair
Substitution
Transposition
Vigenère
XOR
Hash
MAC
Random Number Generator
Signature
Symmetric
AES
ARC4/Spritz
Dragon
IDEA
LFSR
RC6
Algorithms Analysis Visuals Games

Figure 16-2. *The window sections of JCT*

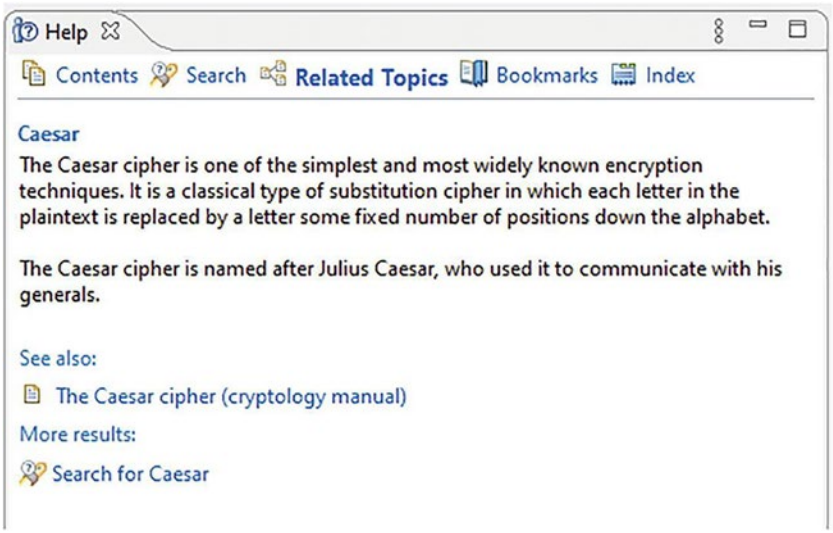

Help
Contents Search Related Topics Bookmarks Index
Caesar
The Caesar cipher is one of the simplest and most widely known encryption techniques. It is a classical type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet.
The Caesar cipher is named after Julius Caesar, who used it to communicate with his generals.
See also:
The Caesar cipher (cryptology manual)
More results:
Search for Caesar

Figure 16-3. *The Help section of JCT*

The first example of using JCT is an easy one; namely, we'll look at the Caesar cipher. To start, edit the text in the third section of the window with a message that will be encrypted using Caesar; then in the Algorithms menu choose Classic ➤ Caesar. Next, a window will open, where you'll choose the secret key and set some settings (Figure 16-4).

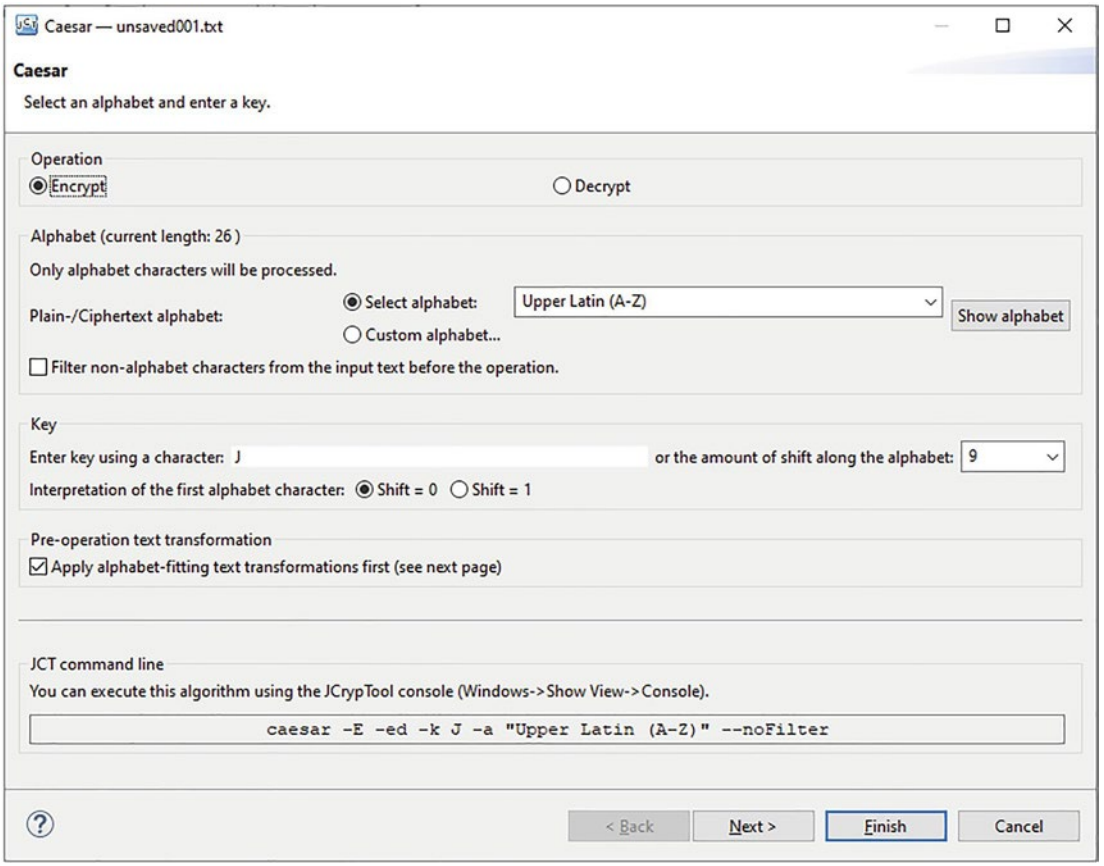

Caesar — unsaved001.txt
Caesar
Select an alphabet and enter a key.
Operation
Encrypt
Decrypt
Alphabet (current length: 26)
Only alphabet characters will be processed.
Plain-/Ciphertext alphabet:
Select alphabet:
Upper Latin (A-Z)
Show alphabet
Custom alphabet...
Filter non-alphabet characters from the input text before the operation.
Key
Enter key using a character: J
or the amount of shift along the alphabet: 9
Interpretation of the first alphabet character: Shift = 0 Shift = 1
Pre-operation text transformation
Apply alphabet-fitting text transformations first (see next page)
JCT command line
You can execute this algorithm using the JCrypTool console (Windows->Show View->Console).
caesar -E -ed -k J -a "Upper Latin (A-Z)" --noFilter
< Back
Next >
Finish
Cancel

Figure 16-4. *Choosing the secret key and the settings for the Caesar cipher*

In the window shown in Figure 16-4, choose the type of operation, i.e., encryption or decryption. You'll see settings for the alphabet. The default setting is Upper Latin (A-Z), but you can choose from a wide range of alphabets; the tool even supports a user-defined alphabet. The alphabet can be displayed by clicking the "Show alphabet" button, and the input can be filtered, such that the symbols that are not contained in the alphabet are excluded from the input. Then select the key by choosing the index of the symbol key in the alphabet or the letter directly. Here also is the setting for the index of

the first character of the alphabet, and it can be set to 0 or 1. The Caesar encryption can be executed from the command line, and in the last setting, the text of the command is shown. Once you've established the encryption key and the settings, then you can click the Next button, which leads to the window in Figure 16-5.

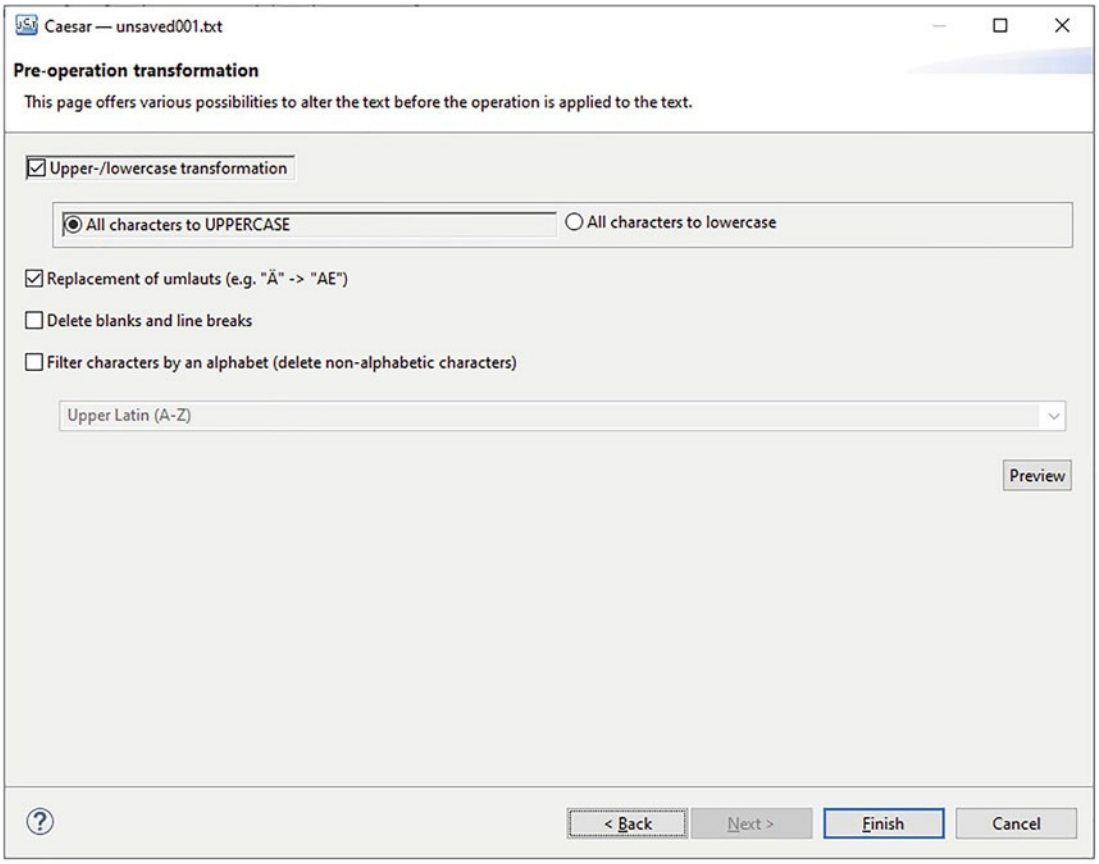

Caesar — unsaved001.txt
Pre-operation transformation
This page offers various possibilities to alter the text before the operation is applied to the text.
Upper-/lowercase transformation
All characters to UPPERCASE
All characters to lowercase
Replacement of umlauts (e.g. "Ä" -> "AE")
Delete blanks and line breaks
Filter characters by an alphabet (delete non-alphabetic characters)
Upper Latin (A-Z)
Preview
< Back
Next >
Finish
Cancel

Figure 16-5. *Choosing the secret key and the settings for the Caesar cipher*

The settings in Figure 16-5 refer to some transformations that can be applied to the input before encryption, for example transforming all characters to uppercase, replacing some special symbols, deleting white spaces, or eliminating the characters that do not belong to the alphabet; then click the Finish button. Figure 16-6, shows the results, which is saved in a text file called `out002.txt` (or a similar name). Note that the encrypted text is displayed in a separate text window. For this example, the text for encryption is "This is an example of using Caesar cipher in JCrypTool."

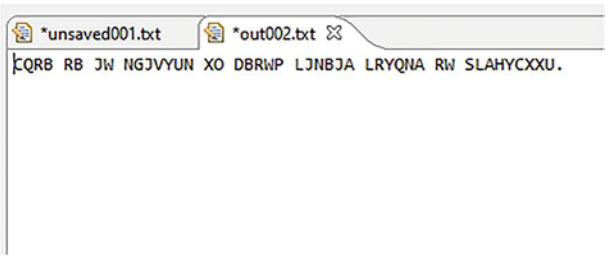


Figure 16-6. *The result of the encryption operation*

The decryption for the Caesar cipher works similarly: from the window shown in Figure 16-4, choose the decryption operation. But before doing that, the text file `unsaved001.txt` should be closed such that the input text for the decryption is the one obtained from the encryption (stored in the text file `out002.txt`). Remember, Caesar is a symmetric cipher; therefore, the decryption key should be the same as the encryption key. In this example, the key is J (or its index in the alphabet; namely, 9 is the starting index is 0). For decryption, the Next button is disabled; therefore, click Finish. Figure 16-7 shows the result for the decryption.

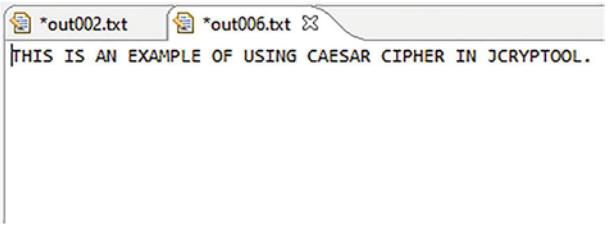


Figure 16-7. *Decryption for the Caesar cipher*

The next example, of the RSA encryption system, is more complex. Before selecting the type of cryptosystem, type the message to be encrypted into a text file. For this example, the message is "Using RSA to encrypt the message." Choose Algorithms ➤ Asymmetric ➤ RSA from the menu. The window shown in Figure 16-8 opens.

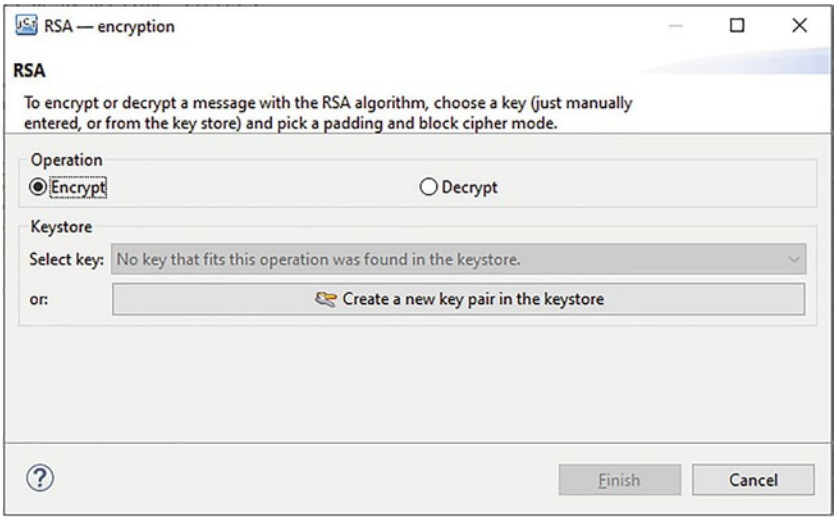

RSA — encryption
RSA
To encrypt or decrypt a message with the RSA algorithm, choose a key (just manually entered, or from the key store) and pick a padding and block cipher mode.
Operation
Encrypt
Decrypt
Keystore
Select key:
No key that fits this operation was found in the keystore.
or:
Create a new key pair in the keystore
Finish
Cancel

Figure 16-8. *The window for RSA encryption*

From here, choose Encrypt as the operation and then click "Create a new key pair in the keystore," if you do not already have a secret-public key pair for the RSA.

For the new key pair, some information should be completed: the contact name, the algorithm for generating the pair and the length, and, lastly, the passwords for the process. The default contact name is [PW: 1234] Alice Whitehead, but for this example, we have changed it to Apress Reader. Keep the default values for the algorithm and key length. The password for this example is RSA@2021. Then click Finish (see Figure 16-9).

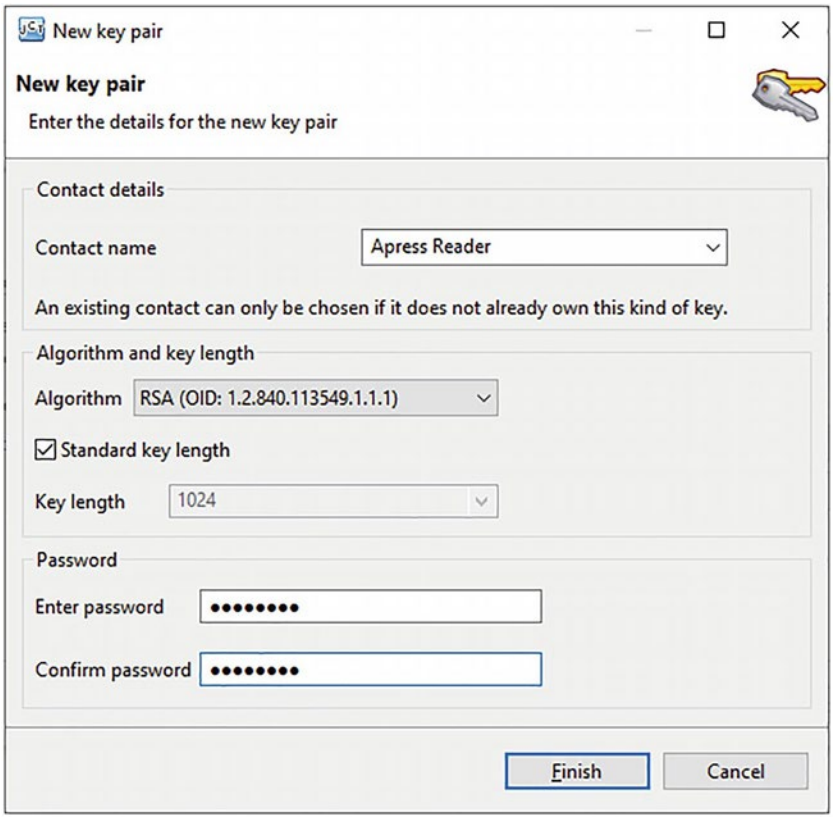


Figure 16-9. *Generating the keys for the RSA encryption*

After generating the keys, you are returned to the window shown in Figure 16-10, where you should click the Finish button.

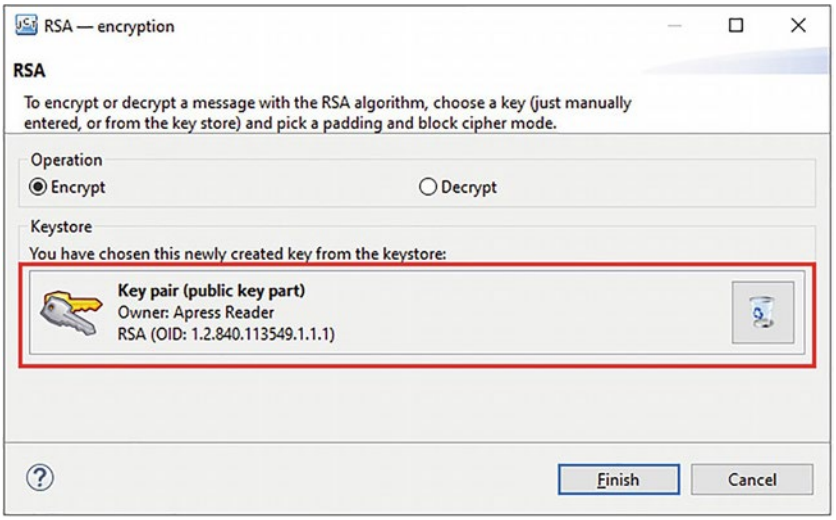

RSA — encryption
RSA
To encrypt or decrypt a message with the RSA algorithm, choose a key (just manually entered, or from the key store) and pick a padding and block cipher mode.
Operation
Encrypt
Decrypt
Keystore
You have chosen this newly created key from the keystore:
Key pair (public key part)
Owner: Apress Reader
RSA (OID: 1.2.840.113549.1.1.1)
Finish
Cancel

Figure 16-10. *RSA window after key pair generation*

The output for the encryption process is the encrypted text, which is shown in a new generated file with the extension `.bin`. The encrypted message from this example looks like the one from Figure 16-11. On the left side of the `.bin` file is the representation of the message in hexadecimal format, while in the right part there is the message in text form. For a selected byte, some information is shown at the bottom of the window: the hexadecimal value, the decimal, and the binary form. At the bottom left, you'll see the number of the current byte and the total number of bytes in the `.bin` file.

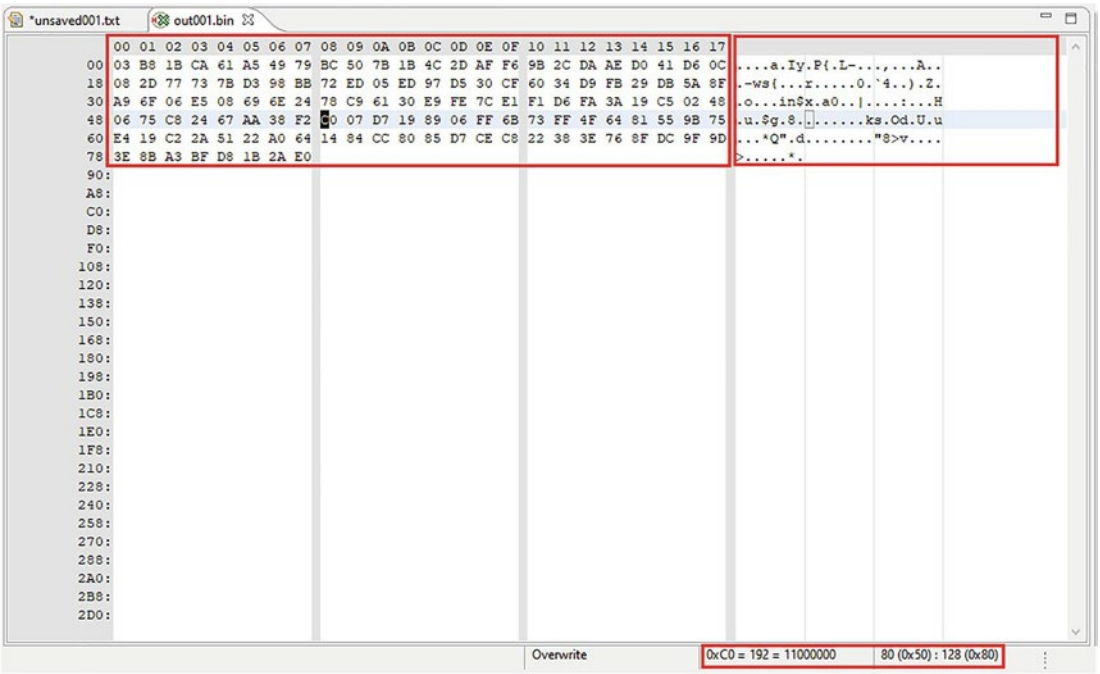


Figure 16-11. *The encrypted message resulted from using the RSA cryptosystem*

To decrypt the message, choose again Algorithms ➤ Asymmetric ➤ RSA. Then, from the open window, choose "Decryption as operation" this time. Note this time the pair of keys previously generated is not preloaded, and the button for generating a new pair is disabled. From here, choose the pair of keys generated previously and then click Finish (Figure 16-12). Next, the password used to create the pair of keys should be introduced; click OK (Figure 16-13). Note it is important to not forget the password used in the key generation.

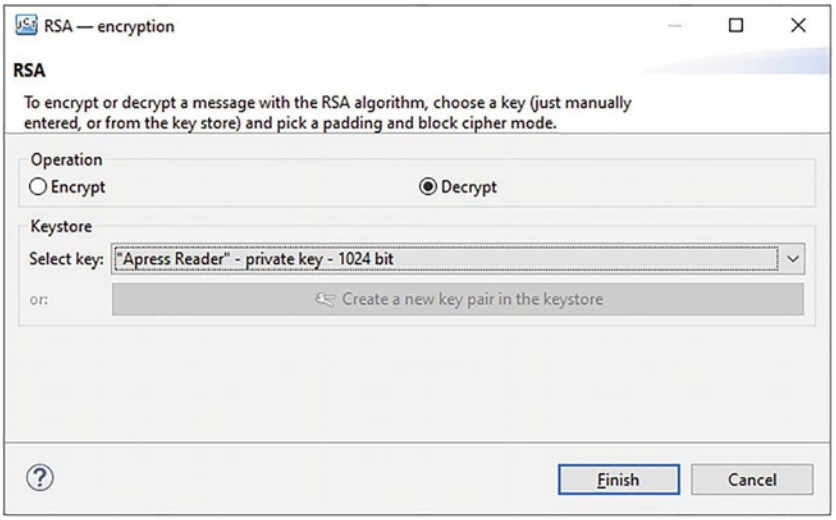

RSA — encryption
RSA
To encrypt or decrypt a message with the RSA algorithm, choose a key (just manually entered, or from the key store) and pick a padding and block cipher mode.
Operation
Encrypt
Decrypt
Keystore
Select key:
"Apress Reader" - private key - 1024 bit
or:
Create a new key pair in the keystore
Finish
Cancel

Figure 16-12. *Decryption for the RSA cryptosystem*

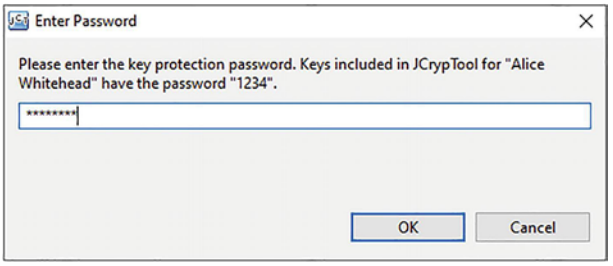

Enter Password
Please enter the key protection password. Keys included in JCrypTool for "Alice Whitehead" have the password "1234".
****
OK
Cancel

Figure 16-13. *Introducing the password for the decryption process*

A similar `.bin` file is generated, which contains the decrypted message in hexadecimal format and text format (Figure 16-14).

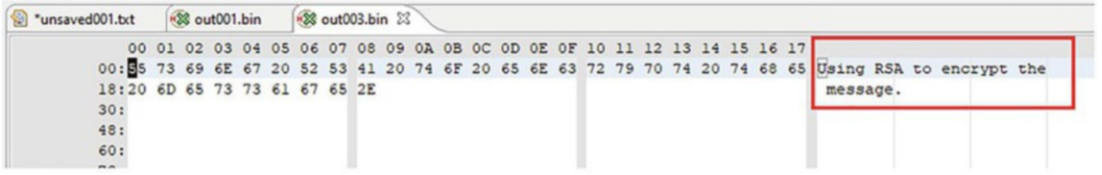


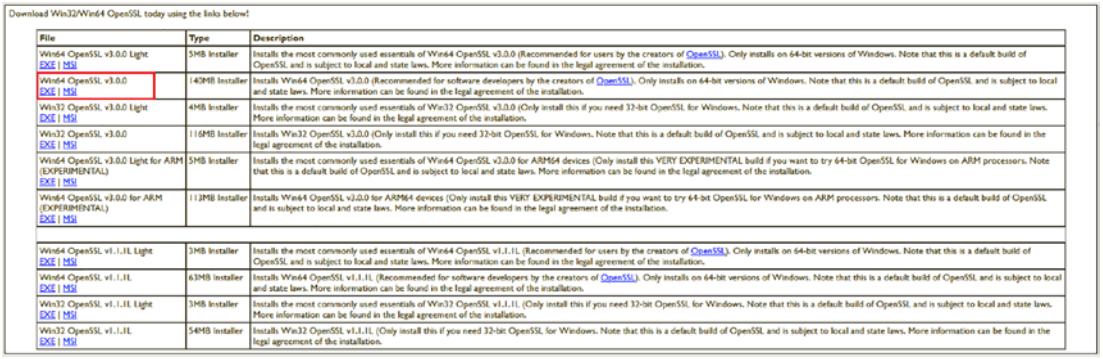
Figure 16-14. *The decrypted message*

CryptTool is a powerful software application for cryptography that can be used for different purposes such as encryption/decryption with a wide range of cryptosystems (symmetric and asymmetric), key generation, hashing, pseudorandom number generation, cryptanalysis, etc.

OpenSSL

OpenSSL is another powerful tool for cryptography. It can be used in different scenarios such as generating keys for different cryptosystems (although the keys can be generated directly with Java) and working with SSL requests. OpenSSL can be downloaded from the official website (`https://slproweb.com/products/Win32OpenSSL.html`). The latest version for Windows at this moment is 3.3.

Download the installer, open it, and then follow the installation steps, leaving the default settings (Figure 16-15). You can download the full version or the lighter version.



Download Win32/Win64 OpenSSL today using the links below!

File	Type	Description
Win64 OpenSSL v3.0.0 Light EXE | MSI	5MB Installer	Installs the most commonly used essentials of Win64 OpenSSL v3.0.0 (Recommended for users by the creators of OpenSSL). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v3.0.0 EXE | MSI	140MB Installer	Installs Win64 OpenSSL v3.0.0 (Recommended for software developers by the creators of OpenSSL). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v3.0.0 Light EXE | MSI	4MB Installer	Installs the most commonly used essentials of Win32 OpenSSL v3.0.0 (Only install this if you need 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v3.0.0 EXE | MSI	116MB Installer	Installs Win32 OpenSSL v3.0.0 (Only install this if you need 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v3.0.0 Light for ARM (EXPERIMENTAL) EXE | MSI	5MB Installer	Installs the most commonly used essentials of Win64 OpenSSL v3.0.0 for ARM64 devices (Only install this VERY EXPERIMENTAL build if you want to try 64-bit OpenSSL for Windows on ARM processors. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v3.0.0 for ARM (EXPERIMENTAL) EXE | MSI	113MB Installer	Installs Win64 OpenSSL v3.0.0 for ARM64 devices (Only install this VERY EXPERIMENTAL build if you want to try 64-bit OpenSSL for Windows on ARM processors. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v1.1.1L Light EXE | MSI	3MB Installer	Installs the most commonly used essentials of Win64 OpenSSL v1.1.1L (Recommended for users by the creators of OpenSSL). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v1.1.1L EXE | MSI	63MB Installer	Installs Win64 OpenSSL v1.1.1L (Recommended for software developers by the creators of OpenSSL). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v1.1.1L Light EXE | MSI	3MB Installer	Installs the most commonly used essentials of Win32 OpenSSL v1.1.1L (Only install this if you need 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.
Win32 OpenSSL v1.1.1L EXE | MSI	54MB Installer	Installs Win32 OpenSSL v1.1.1L (Only install this if you need 32-bit OpenSSL for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.

Figure 16-15. *Official download for OpenSSL*

The next step is to add the OpenSSL configuration and binaries to the system variables.

To run Environment Variables, go to System Properties and click Environment Variables (Figure 16-16). From the opened window, click the New button at the bottom (Figure 16-17), and complete the information for the OpenSSL configuration as in Figure 16-18.

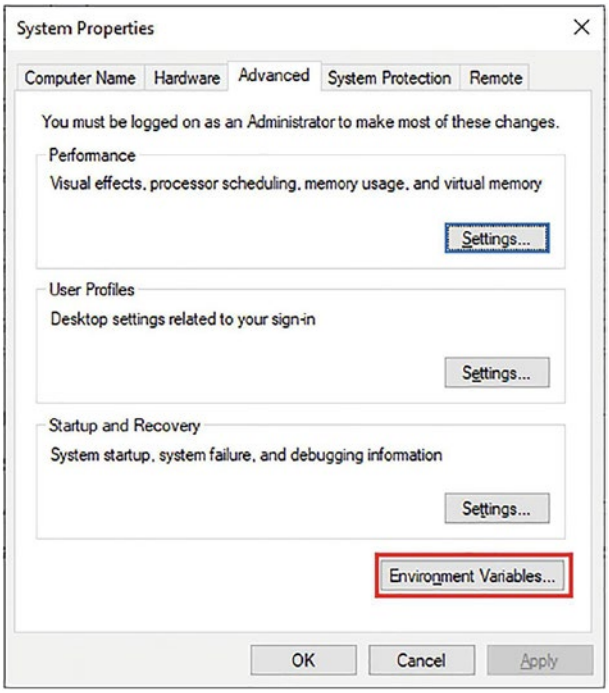


Figure 16-16. *Opening the system variables for editing*

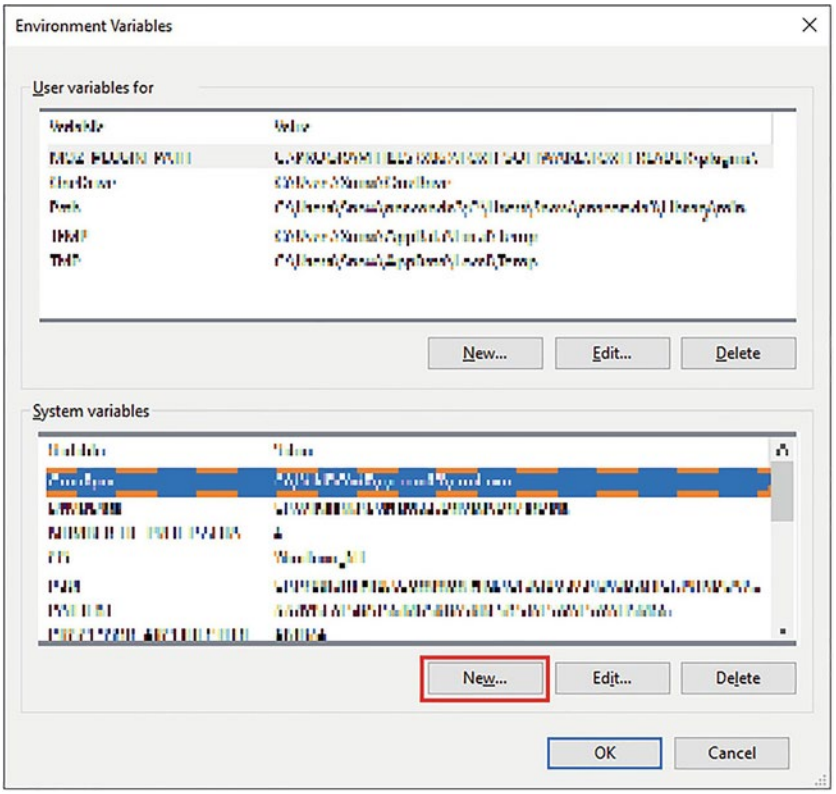


Figure 16-17. *Adding a new entry*

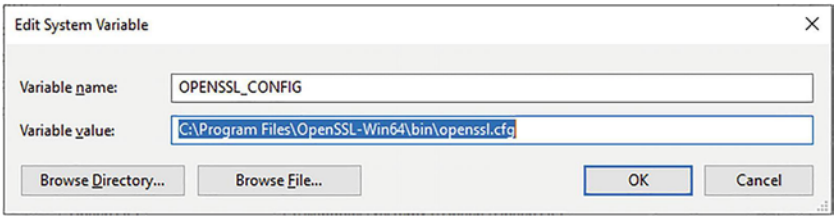


Figure 16-18. *The configuration values for the new entry*

Then, add the binary files for OpenSSL, as follows: in the "System variables" section select Path and then Edit (Figure 16-19). Here, click New and paste the path of the `bin` folder for OpenSSL (Figure 16-20). Click OK, then OK again, and OK again. The System Properties window closes.

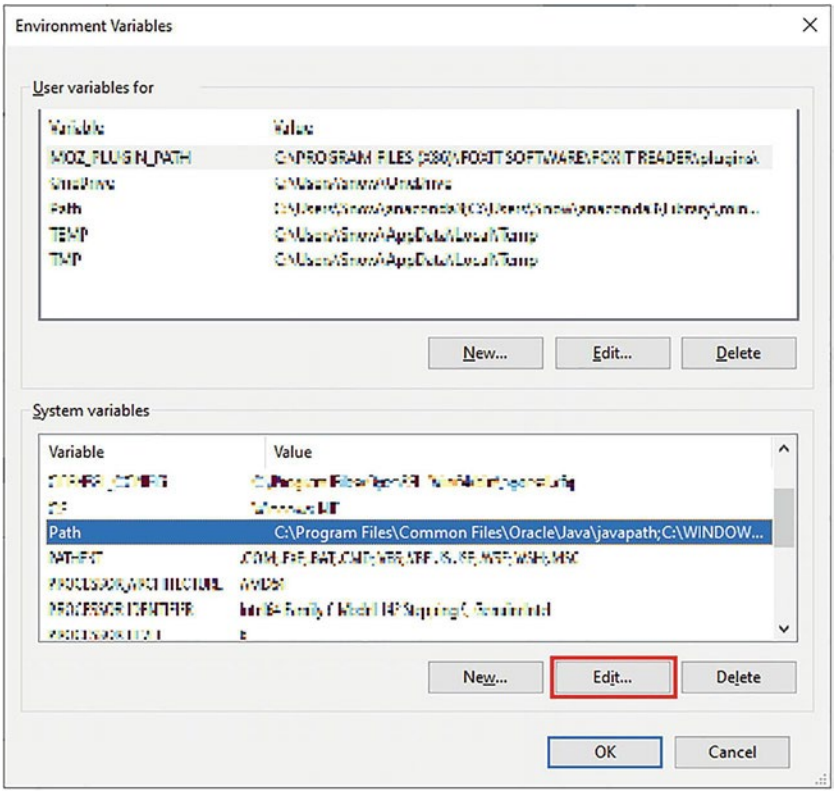


Figure 16-19. *Editing the path from System Variables*

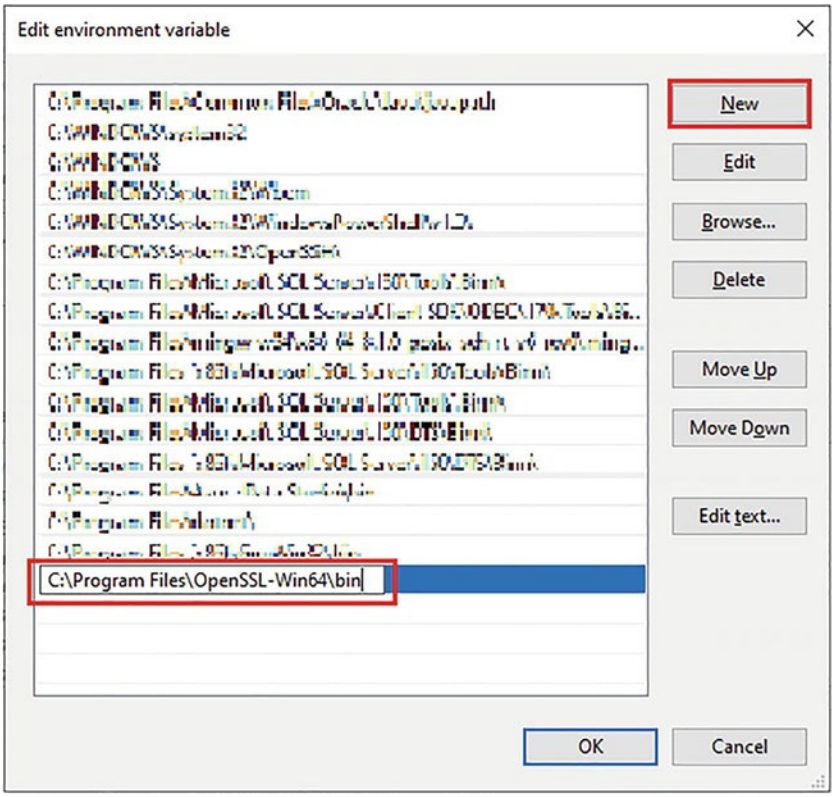


Figure 16-20. *Adding the bin folder for OpenSSL to Path*

To test whether OpenSSL was configured and added properly, open a terminal window and type the command `openssl version`. The version of the library should be displayed (Figure 16-21). If you need help with the OpenSSL commands, just type the `openssl` command in the terminal.

Administrator: Command Prompt
C:\Program Files\OpenSSL-Win64\bin>openssl version
OpenSSL 3.3.0 9 Apr 2024 (Library: OpenSSL 3.3.0 9 Apr 2024)
C:\Program Files\OpenSSL-Win64\bin>

Figure 16-21. *Testing OpenSSL*

To demonstrate the use of OpenSSL, the next example shows how to generate the private and public keys for the RSA cryptosystem.

First, change the current directory to a specific folder that can be easily found. In this example, we are placing the folder in `C:\openssl`. The command to generate Alice's private key is as follows:

```
openssl genrsa -out alicePrivKey.pem 2048
```

To generate the public key, the command is as follows:

```
openssl rsa -pubout -in alicePrivKey.pem -out alicePublicKey.pem
```

Check the content of the current directory; indeed, the two keys have been generated (Figure 16-22).

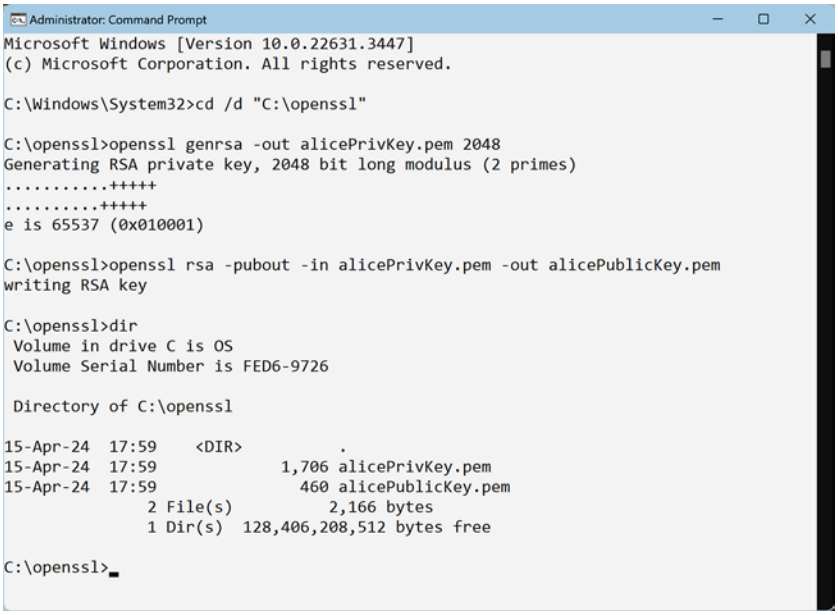

Administrator: Command Prompt
Microsoft Windows [Version 10.0.22631.3447]
(c) Microsoft Corporation. All rights reserved.
C:\Windows\System32>cd /d "C:\openssl"
C:\openssl>openssl genrsa -out alicePrivKey.pem 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
C:\openssl>openssl rsa -pubout -in alicePrivKey.pem -out alicePublicKey.pem
writing RSA key
C:\openssl>dir
Volume in drive C is OS
Volume Serial Number is FED6-9726
Directory of C:\openssl
15-Apr-24 17:59 <DIR> .
15-Apr-24 17:59 1,706 alicePrivKey.pem
15-Apr-24 17:59 460 alicePublicKey.pem
2 File(s) 2,166 bytes
1 Dir(s) 128,406,208,512 bytes free
C:\openssl>

Figure 16-22. *Generating the keys for the RSA cryptosystem*

Integrating OpenSSL with Java

Further, we will provide an example of integrating OpenSSL commands into Java projects. When combined with Java, OpenSSL can be particularly useful for managing cryptographic tasks such as certificate management, key generation, and encryption/decryption processes. The following are several practical use cases for OpenSSL with Java:

- **Generating Keys and Certificates.** OpenSSL can be used to generate keys and certificates that Java applications can utilize for SSL connections or for signing and verifying data.
- **Encryption and Decryption.** OpenSSL provides a wide range of encryption algorithms that can be used for encrypting data outside of Java and then decrypting it within a Java application, or vice versa.
- **Certificate Conversion.** OpenSSL can convert certificates into different formats that are compatible with Java's KeyStore requirements.

To integrate OpenSSL with Java, the `Runtime.exec()` method in Java can be used to run OpenSSL commands. The output from these commands can then be captured and utilized within the Java application. We have already seen in Figure 16-22 the commands that generate private and public keys for the RSA cryptosystem. Listing 16-1 shows how to integrate these commands into a Java application, while Figure 16-23 shows the output for Listing 16-1.

Listing 16-1. Example of Integrating OpenSSL Commands in Java Application

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class ExampleOpenSSL {
    public static void main(String[] args) {
        try {
            // Generate Bob's private key
```

```
10              String priv_key_cmd = "openssl genrsa -out BobPrivKey.
                pem 2048";
11              executeCommand(priv_key_cmd);
12
13              // Extract Bob's public key
14              String pub_key_cmd = "openssl rsa -pubout -in BobPrivKey.
                pem -out BobPubKey.pem";
15              executeCommand(pub_key_cmd);
16
17              // Print Bob's public key to console
18              BufferedReader br = new BufferedReader(new InputStream
                Reader(new FileInputStream("BobPubKey.pem")));
19              String l;
20              while ((l = br.readLine()) != null) {
21                  System.out.println(l);
22              }
23              br.close();
24
25          } catch (Exception e) {
26              e.printStackTrace();
27          }
28      }
29
30      private static void executeCommand(String cmd) throws IOException,
        InterruptedException {
31          Process proc = Runtime.getRuntime().exec(cmd);
32          proc.waitFor();
33          BufferedReader br = new BufferedReader(new
            InputStreamReader(proc.getInputStream()));
34          String l;
35          while ((l = br.readLine()) != null) {
36              System.out.println(l);
37          }
38      }
39  }
```

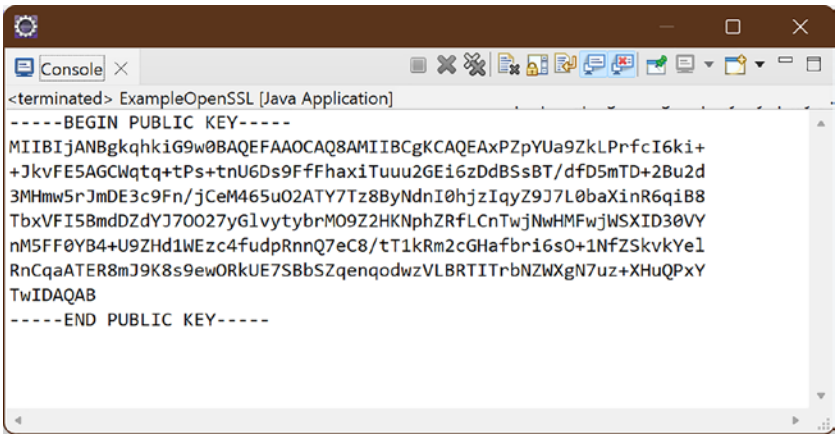


Figure 16-23. *The content of the public key*

Listing 16-1 shows how to utilize OpenSSL to generate a private and public RSA key pair and subsequently read and display the public key using standard Java I/O operations. Initially, the code sets up commands for OpenSSL to generate a 2048-bit RSA private key and to extract the corresponding public key from the generated private key file. These commands are passed to a helper method, `executeCommand`, which executes them using Java's `Runtime.exec()` method. This method runs the command as a separate process, waits for it to complete, and reads any output from the command via a `BufferedReader` attached to the process' input stream, printing it line by line to the console. In the main method, after executing the OpenSSL commands, another `BufferedReader` is set up to read from the newly created public key file (`BobPubKey.pem`). It iterates over each line of the file, printing the public key to the console and ensuring that the cryptographic key is correctly generated and accessible. The reading process continues until all lines are consumed, with each line displayed on the console, thus verifying the public key's content. The code is designed to handle exceptions such as `IOException` or `InterruptedException` by catching them and printing a stack trace, which aids in debugging potential issues during execution. Integrating OpenSSL with Java shows a practical approach to managing cryptographic operations in Java applications based on external cryptographic tools for enhanced security functionalities. After running the code, two files will be created in the project's folder (if not specified otherwise with another path), namely, `BobPrivKey.pem` and `BobPubKey.pem`, corresponding to the private and public keys.

However, when integrating different technologies, especially for security purposes, these best practices should be considered:

- **Security Considerations.** When integrating OpenSSL with Java, ensure that any sensitive data handled by commands is secured, and consider the security implications of exposing cryptographic material through the command line.
- **Error Handling.** Properly handle errors from OpenSSL commands in the Java code, including checking the exit value of the process.
- **Resource Management.** Ensure all resources such as streams and processes are properly closed after their use to prevent resource leaks.

Conclusion

Cryptography and information security are some of the most exciting and important branches of computer science and information technology as technology evolves and threats may appear anytime, even from unexpected sources.

This chapter presented two important software applications and tools used in these fields, namely, CrypTool and OpenSSL, and showed how OpenSSL commands can be integrated into Java applications. Both CrypTool and OpenSSL are very powerful and provide complex operations. Also, the chapter presented how the tools can be installed and how they can be used. The JCrypTool version is even more relevant as it is written in Java, while OpenSSL can be used to parametrize things in Java cryptosystems or for other cryptographic purposes.

References

[1] CrypTool. Available online: https://www.cryptool.org/en/

[2] OpenSSL. Available online: https://slproweb.com/

Index

A

Access control, 56, 57, 219
Advanced electronic signatures (AdES), 243
Advanced encryption schemes
  homomorphic encryption, 205–209
  SE, 210–212
Advanced Encryption Standard (AES), 15, 65
  advantages, 175
  cryptographic key, 263, 264
  development, 166
  disadvantages, 176
  encryption and decryption, 176, 180
  encryption process, 181
  operation modes
    CBC, 176
    CFB, 177
    counter (CTR), 177
    ECB, 176
    GCM, 177
    implementation, 177–180
    OFB, 177
  padding operations, 175
  protocols, 166
  supports key, 166
  symmetric block cipher, 175
AES_Cryptosystemn, 33
Algebraic structures, 126, 127, 199
Alice and Bob, 108–111, 113
ANSI X9.18-1985 Appendix C, 137
ANSI X9.31-1998 Appendix A.2.4, 138
ANSI X9.62-1998 Annex A.4, 138
Application programming interfaces (APIs), 9, 10, 19, 23, 29, 55, 56, 60, 238
Applied cryptography
  authentication and identity verification, 5
  cryptographic algorithms, 3
  cryptography theoretical principles, 4
  data security, 4
  digital signatures, 5
  E-commerce and online transactions, 5
  information security, 6
  integrity of information, 5
  Internet and network security, 5
  Knowledge is power, 6
  national security and defense, 5
  nonrepudiation, 5
  practical implementation, 6
  practical security, 6
  privacy, 5
  system integration, 3
  update and maintenance, 4
  user interface and experience, 4
Asymmetric encryption schemes, 14
  concept, 183
  cryptosystems, 184
  digital signatures, 184

© Stefania Loredana Nita, Marius Iulian Mihailescu 2024
S. L. Nita and M. I. Mihailescu, *Cryptography and Cryptanalysis in Java*,
https://doi.org/10.1007/979-8-8688-0441-0

Asymmetric encryption schemes (*cont.*)
ECC, 199
ElGamal, 191–197
enabling secure communication, 184
Merkle-Hellman cryptosystem, 197, 198
PQC, 200
RSA, 185–191
secure key distribution, 184
sender (*S*) and receiver (*R*), 185
Atmospheric noise, 135
Authorities and third parties, 118

B

BB84 protocol
components, 109, 110
QKD, 109
security based, quantum principles, 110, 111
simulation, 111, 113
Bell state, 102, 104, 114
Bennett 1992 (B92) protocol, 113
Beth-Piper PRNG, 141
BigInteger, 122–124, 189, 190
Bits, 117
BLAKE family, 154
Blockchain, 155, 156, 221, 235
blockDecryption(String encryptedBlock) method, 96
blockEncryption(String plainBlock) method, 95
Blum-Blum-Shub (BBS), 139, 140
Blum integer, 124, 131, 139
Blum-Micali PRNG, 141
Brute-force attack, 83, 152, 165
byteToString method, 189

C

Caesar cipher, 77
cryptanalysis, 83
first century BC, Roman Empire, 78
implementation, 79, 80, 82, 83
k = 6, 78, 79
Rot13, 78
substitution cipher, 78
Calculational SVP, 249
Chosen ciphertext attacks (CCA), 191, 197
Cipher block chaining (CBC)
AES, 176
DES, 170
Cipher feedback (CFB)
AES, 177
DES, 171
Ciphertext, 8, 65, 83, 171, 177
Classical cryptography, 11, 12, 183
Closest vector problem (CVP), 248
Cloud-native Java application development, 10
Cloud-native technologies, 40
Collision-resistant, 152
Complementary variables, 105, 107
Conditional probability, 120
Context-Specific Deserialization Filters (JEP415), 34
Continuous variable QKD, 114
Counter (CTR)
AES, 177
DES, 171
Cross-Origin Resource Sharing (CORS), 42
Cross-Site Request Forgery (CSRF), 42
attack, 42
prevent CSRF attacks, 44–47
vulnerability, 44
Cryptanalysis, 83, 89

definition, 8
goal, 119
Cryptanalytic attacks, 158–159
CryptoExplorer, 270
Cryptographic algorithms, 3
Cryptographic Algorithm Validation Program (CAVP), 130
Cryptographically secure pseudorandom number generators (CSPRNGs), 135–137, 139, 146–147
Cryptographic functions, 55, 99, 152, 200
Cryptographic Module Validation Program (CMVP), 130, 131
Cryptographic operations, 24, 26, 35, 55, 63
Cryptographic primitive, 9, 118, 130, 158, 160, 209, 270
Cryptographic service provider (CSP), 59
Cryptography, 6, 185
definition, 8
Enigma machine, 7
field, 127
formal techniques, 12
group, 126
history, 7
polyalphabetic substitution (Middle Ages), 7
substitution approach, 7
theoretical (*see* Theoretical cryptography)
tools, 17
Cryptology, definition, 8
CrypTool (CT)
.bin file, 277
Caesar cipher, 272, 273
CryptoExplorer, 270
decrypted message, 279
decryption for Caesar cipher, 274
encryption operation, 274
File Explorer, 270
JCT, 269–272
out002.txt, 273
RSA
cryptosystem, 278, 279
encryption system, 274–276
window after key pair generation, 276, 277
secret key, 272, 273
.txt file, 270
type of operation, 272
Cyclic group, 127, 191, 192

D

Data encryption, 65–67
Data Encryption Standard (DES), 15
block cipher, 165, 166
vs. brute-force attack, 165
development, 165
eliminating unnecessary file existence check, 175
encryption and decryption, 169–171, 173, 174
encryption process, 181
generation of keys, 168, 169
improved error handling, 175
LUCIFER algorithm, 166
operation modes
CBC, 170
CFB, 171
Counter (CTR), 171
ECB, 170
implementation, 171–174
OFB, 171
refactoring file processing, 175
64-bit plaintext, 167

Data Encryption Standard (DES) (*cont.*)
TDEA/3DEA, 167, 168
TDES, 167
3DES, 166
try-with-resources, 174
weaknesses, 165
Data integrity, 15, 53, 67, 137, 155, 159, 220, 233
Data security, 4, 109, 184
Decentralized identities (DIDs), 221
Decisional SVP, 249
Definitions
attacks, 119
authorities and third parties, 118
bits, 117
cryptographic primitive, 118
cryptosystem, 118
hardness assumption, 118
protocol, 118
sets, 117
DESCHALL Project, 165
DHPrivateKey, 58
DHPublicKey, 58
Differential phase shift (DPS) QKD, 114
Digital banking
online banking authentication, 219
transaction signing, 220
Digital fingerprint of files, 159
Digital Signature Algorithm (DSA), 16, 137, 234, 235
Digital signatures, 5, 15, 69, 128, 137, 154, 184
data integrity, 233
description and security requirements, 234, 235
ElGamal, 238–242
implementations, 235–238
key generation, 233, 239
message
authentication, 233
signing, 239
verifying, 239
nonrepudiation, 233
signing, 233
standards and regulations, 242, 243
verifying, 234
Digital Signature Standard (DSS), 234
Direct cryptanalytic attacks, 142
Discrete logarithm problem (DLP), 129, 141, 191, 196, 238
Discrete sample space, 119
Distributed ledger, 155

E

E91 (Ekert protocol), 113
E-commerce and online transactions, 5
Edwards-Curve signature algorithm, 235
Electronic codebook (ECB)
AES, 176
DES, 170
Electronic communication, 1
Electronic Identification and Trust Services for Electronic Transactions in the Internal Market (eIDAS), 243
Electronic Signatures in Global and National Commerce Act (ESIGN), 242
ElGamal, 238
chosen ciphertext attacks, 197
cryptosystem, 192, 193
decryption algorithm, 192
discrete logarithm problem, 196
DLP, 191
encryption algorithm, 192

homomorphic properties, 196
implementation, 193–196
key generation algorithm, 192
malleability, 196
shared secret, 192
signature scheme, 239–243
Elliptic-curve cryptography (ECC), 128, 129, 138, 185, 199
Elliptic Curve Digital Signature Algorithm (ECDSA), 16, 234
Elliptic Curve Discrete Logarithm Problem (ECDLP), 199
Elliptic curves, 128, 129
Encrypted communication, 42
Encrypting messages methods, 77
Engine classes, 59, 63
Enigma machine, 6, 7
Exclusive-or (XOR) operations, 140

F

Fast bootstrapping, 209
Federal Information Processing Standards (FIPS), 130, 131, 234
Feedback function, 140
Feige-Fiat-Shamir (FFS) identification protocol
actions, 224
implementation, 224–230
keys generation, 223
messages exchange, 224
rejecting the identity, 230, 231
security properties, 231
system parameters, 223
Feistel block cipher algorithm, 166
FHE libraries, 208
File Explorer, 270
FIPS 186-4, 137
FIPS regulations, 250
Foreign Function & Memory API (Incubator) (JEP 412), 20, 21, 23, 25, 34, 35
Fully homomorphic encryption (FHE), 205, 207–209, 212, 213, 247

G

Galois and counter mode (GCM), 177
Galois fields, 127
Geffe generator, 141
Generic attacks, 158
getDefault() method, 142
GGH cryptosystem, 247
Gollmann PRNG, 141
GOST hash function, 154

H

Hardness assumption, 118, 122, 124, 129, 130, 185, 190, 191, 209, 231, 232, 248
Hardware random number generators, 135
Hash functions, 14, 67–69, 243
applications, 155
BLAKE family, 154
blockchain, 155, 156
collision-resistant, 152
collisions, 151
cryptanalytic attacks, 158
digital fingerprint of files, 159
digital signatures, 154
generic attacks, 158
GOST, 154
hashing class, 151
hash value, 152

Hash functions (*cont.*)
indifferentiability, 158
indistinguishability, 158
keyed cryptographic, 152, 153
MD, 154, 155
message digest, 151
near-collision, 158
noninvertibility, 152
noninvertible, 151
properties, 158
RIPEMD family, 155
SHA-256, 159, 160
SHA-512, 152, 156, 157
SHA family, 155
start-collision, 158
structure of chain, 156
unkeyed cryptographic functions, 152–154
weak collisions, 151, 152
whirlpool, 155
Hashing class, 151
Hash value, 14, 64, 67, 151, 152, 154–156, 158, 159, 242
Heisenberg Uncertainty Principle, 105–108, 110
HElib, 208
Hill cipher, 78
cryptanalysis, 96
decrypt and encrypt the message, 90
implementation, 91, 95, 96
in 1929, 89
linear algebra, 89
Homomorphic encryption (HE), 190
evaluation, 206
FHE, 207–209
history, 207
homomorphism, 206
PHE, 207
properties, 206
structures, 206
SWHE, 207
unpadded RSA, 206, 207
use case, 205
Homomorphism, 190, 206

I

Identification schemes
access control, 219
blockchain identities, 221
completeness, 223
digital banking, 219, 220
FFS identification protocol, 223–231
IoT device authentication, 220
message exchange, 222
OTPs, 222
passwords, 221
PINs, 222
prover, 219
rudimentary technique, 219
secure online voting, 220
something inherited, 221
something known, 221
something possessed, 221
soundness, 223
two-step authentication, 222
types of attacks, 221
verifier, 219
weak authentication, 221
zero-knowledge proofs, 222, 223
Identification techniques, 15
Indifferentiability, 158
Indistinguishability, 158
Initial permutation (IP) function, 170
Input-based attacks, 142

Insecure classical channel, 109
Interfaces and main classes, 64
IoT device authentication, 220–221

J

Jakarta CDI, 51
Jakarta EE 11, 10
  aims, 41
  authentication mechanisms, 47
  CDI
    advantage, 52
    components, 51
  cloud-native technologies, 40
  community and ecosystem, 41
  compatibility with Java SE features, 40
  criteria and consistency, 41
  future, 52
  future vision, 41
  goal, 39
  JAAS, 53
  Java, cloud-native application development, 39
  JCE, 53
  JSSE, 53
  modularity and microservices, 40
  performance and efficiency, 40
  prevent CSRF attacks, 44–47
  security, 41
    annotations, 47
    improvements, 41, 42, 44
    measures, 40
    strategy, 47
  user-friendly authentication modules, 49
  web profile, 48
    business applications, 49, 50
Jakarta EE's development, 10
Jakarta Security, 39, 47, 49, 51, 52
Jakarta Security 1.0, 50
Java 21, PRNGs, 27
Java Authentication and Authorization Service (JAAS), 53
Java Cryptography Architecture (JCA), 10, 11
  access control policies, 56
  APIs, 56
  architecture and design principles, 56–62
  authentication, 56
  cryptographic functions, 55
  cryptography operations, 75
  JDK, 55
    components, 57, 58
  secure communication, 55
Java Cryptography Extension (JCE), 53, 57, 176
Java Development Kit (JDK), 10, 19, 55
Java EE 8, 39
Java Language Specification (JLS), 19
Java platform, 19, 55
Java Secure Socket Extension (JSSE), 53, 56
java.security, 55, 57
java.security.Provider, 59
java.security.SecureRandom, 28, 144–146
Java Standard Edition (Java SE), 19
java.util.Random, 13, 28
java.util.Random Class, 142–146
Java Virtual Machine (JVM), 9, 34
Java Virtual Machine Specification (JVMS), 19
javax.crypto, 57, 58
javax.crypto.interfaces, 58
javax.crypto.spec, 58

JCA classes and algorithms
data encryption, 65
digital signatures
advantages, 69
generating signatures, 70, 72
verify different aspects, 69
verifying signature process, 72, 74
encryption process, output, 67
engine class, 63
hash functions, 67–69
interfaces and main classes, 64
JCrypTool (JCT), 269–272
JCrypTool.exe, 269
JDK 17, 19, 20
JDK 21, 9
Context-Specific Deserialization Filters (JEP415), 34
cryptographic operations
Foreign Function & Memory API (Third Preview) (JEP 442), 25
Pattern Matching for switch (Fourth Preview) (JEP 433), 24
Record Patterns (JEP 440), 25
sealed types (JEP 409), 26
Virtual Threads (JEP 444), 26
Eclipse environment, 21
Foreign Function & Memory API (Incubator) (JEP412), 34
Foreign Function & Memory API (Third Preview) (JEP 442), 21, 23
Key Encapsulation Mechanism API (JEP 452), 29
new features, 19, 35
Pattern Matching for switch (Fourth Preview) (JEP 433), 21, 22, 29–32
record patterns (JEP 440), 21, 22
Sealed Classes (JEP 409), 32, 33
Sealed types (JEP 409), 23
Structured concurrency (JEP 453), 21
Vector API (Second Incubator) (JEP414), 34
Virtual threads (JEP 444), 21, 23
JEP 409: Sealed Types, 23, 26
JEP 433: Pattern Matching for switch (Fourth Preview), 21, 22, 24, 29–32
JEP 440: Record Patterns, 21, 22, 25
JEP 442: Foreign Function & Memory API (Third Preview), 21, 23, 25
JEP 444: Virtual Threads, 21, 23, 26
JEP 452. Key Encapsulation Mechanism API, 29
JEP 453: Structured Concurrency, 21
JSON Web Token (JWT), 39, 45, 46

K

Key distribution problem, 183
Keyed cryptographic hash functions, 152, 153
Key Encapsulation Mechanism API (JEP 452), 29
Knapsack approach, 197

L

Lattice-based cryptography, 16
audit and compliance, 267
CVP, 248
definition, 248
fully homomorphic encryption schemes, 247
key exchange mechanism, 266
key management, 267
in $\mathbb{R}^2$, 248
regular updates, 267
SBP, 248

secure application development, 266
SIVP, 248
SVP, 248, 249
Left Plain Text (LPT), 170
Legacy category, 28
Linear circuit PRNGs, 140–141
Linear congruential generator (LCG), 138, 139
Linear congruential PRNGs, 139
Linear feedback shift register (LFSR), 140
LUCIFER algorithm, 166

M

MAC algorithms, 56
Malleability, 196
Measurement-device independent QKD (MDI-QKD), 114
measureMomentum() method, 107
measurePosition() method, 107
Merkle-Hellman cryptosystem
decryption and encryption algorithm, 198
key generation algorithm, 198
knapsack approach, 197
subset sum problem, 198
Message authentication, 233
Message Authentication Code (MAC) algorithm, 55
messageDecryption(String encryptedMessage) method, 96
MessageDigest, 59
Message digest (MD) hash functions, 151, 154, 155
messageEncryption(String plainMessage), 96
Message exchange, 118, 222, 224, 231
Microservices, 40, 45, 266
Modern encryption algorithms, 65
Module lattices, 209
Monte Carlo algorithms, 141
Multifactor authentication (MFA), 219
Multi-keyword searchable encryption schemes, 212

N

National Institute of Standards and Technology (NIST), 130, 165, 166
competition, 200
Randomness Tests (Special Publication 800-22), 131
SP 800-90A, 137
National Security Agency (NSA), 155, 165
Near-collision, 158
NIST test suites
CAVP, 130
NIST Randomness Tests (Special Publication 800-22), 131
Transitioning to Post-Quantum Cryptography, 130
No-cloning theorem, 110, 115
Nonrepudiation, 5, 69, 184, 220, 233
NP-complete, 152
Nth Degree Truncated Polynomial Ring Units (NTRU), 16
DecryptTheMessage() method, 261, 264
decryption, 249, 260
EncryptTheMessage() method, 261–264
encryption, 16, 249, 260
ExecuteNTRUEncryption DecryptionProcess() function, 260–262
key generation, 249

Nth Degree Truncated Polynomial Ring Units (NTRU) (*cont.*)
library, 250–265
operations, encryption, and decryption, 253–260
parameters, 264
secure application development, 266, 267
SVP, 249

O

One-time pad (OTP), 86
One-time passwords (OTPs), 146, 147, 222
Online banking authentication, 219
OpenSSL
adding a new entry, 280, 282
binaries and configuration, 280
binary files, 283
bin folder, 283, 284
BobPrivKey.pem, 288
BobPubKey.pem, 288
BufferedReader, 288
certificate conversion, 286
configuration, 280, 282
encryption and decryption, 286
error handling, 289
executeCommand, 288
generating keys and certificates, 286
InterruptedException, 288
IOException, 288
with Java, 286–289
official download, 280
openssl version, 284
resource management, 289
RSA cryptosystem, 285
Runtime.exec() method, 286, 288
security considerations, 289
System Properties and Environment Variables, 280, 281
system variables, 283
testing, 284
Operation modes
AES
CBC, 176
CFB, 177
counter (CTR), 177
ECB, 176
GCM, 177
OFB, 177
DES
CBC, 170
CFB, 171
counter (CTR), 171
ECB, 170
OFB, 171
Optimal Asymmetric Encryption Padding (OAEP), 191
Order of a group, 127
OutOfMemoryError, 124
Output feedback (OFB)
AES, 177
DES, 171

P

Padding operations, 175
PALLISADE, 209
Partial homomorphic encryption (PHE), 207
Password-Based Key Derivation Function 2 (PBKDF2), 180
Peikert's RLWE Key-Exchange, 247
Permutations, 168, 169
PINs, 222

Plaintext, 65, 85, 89, 168, 170, 171, 176, 177, 180, 189, 206, 221, 243, 263, 264
Point addition, 129
Pollard-Rho attack, 129
Polyalphabetic substitution, 7, 78, 84
Post-quantum cryptography (PQC), 130, 200, 247
pow() method, 124
Practical security, 6
Prime factorization, 122, 130
PRNGs, Java 21, 27
Probabilities and statistics
  conditional probability, 120
  entropy, 121
  event, 119
  experiment, 119
  random variable, 120
  sample space, 119
  security analysis process, cryptosystems, 119
Pseudorandom generators (PRGs), 13, 35, 147
Pseudorandom number generators (PRNGs)
  ANSI X9.18-1985 Appendix C, 137
  ANSI X9.31-1998 Appendix A.2.4, 138
  ANSI X9.62-1998 Annex A.4, 138
  BBS, 139, 140
  conditions, 136
  FIPS 186-4, 137
  Java.util.Random Class, 142–146
  LCG, 138, 139
  linear circuits, 140, 141
  mathematical problem, 141
  next-bit test, 136
  NIST SP 800-90A, 137
  random bits, 136, 137
  randomness degree, 137
  Ranrot class, 141
  RSA problem, 141
  security, 142
  seed, 136
  state compromise extensions, 136
Public-key cryptography (PKC), 128, 183, 185, 199, 221, 233
Public-key encryption systems, 185, 190
Public key infrastructure (PKI) systems, 56
Public-key schemes, 233
Public-key searchable encryption (PKSE), 211, 212

Q

Quadratic residue modulo *n*, 126
Qualified electronic signatures (QES), 243
Quantum and classical channels, 109
Quantum bit/qubit, 99, 100, 109
Quantum computing, 12, 109, 115, 130, 200, 234
Quantum cryptography, 12
  B92 (Bennett 1992 protocol), 113
  BB84 protocol, 109–113
  challenges, 109
  continuous variable QKD, 114
  DPS QKD, 114
  E91 (Ekert protocol), 113
  Heisenberg Uncertainty Principle, 105, 107
  MDI-QKD, 114
  QKD, 108, 109
  QSDC, 114
  quantum entanglement, 102, 105
  quantum physics principles, 99
  quantum superposition, 99–101
  Six-State protocol, 114
  TF-QKD, 114

Quantum entanglement, 102–105, 114
Quantum key distribution (QKD), 12
  challenges, 109
  eavesdropping detection, 108
  quantum cryptography, 108
  security, 108
  structure, 108
Quantum particle, 105
Quantum random number generators, 135
Quantum-resistant, 11, 115, 208, 209, 249
Quantum secure direct communication (QSDC), 114
Quantum state, 102–104, 108, 110, 113, 114
Quantum superposition, 99–101, 115
Quantum systems, 12, 99, 105
Qubits
  encoding information, 109
  initialization, 101
  measurement, 101, 110
  simulation, 100, 101

R

The RACE Integrity Primitives Evaluation Message Digest (RIPEMD) family, 155
Random bits, 136–138
Randomly generated numbers, 135
Randomness, 13, 131, 135–137, 147
Random numbers, 135, 142
Random variable, 120, 121
Record patterns (JEP 440), 22, 25, 35
Residual generator, 139
Right Plain Text (RPT), 170
Ring switching, 209
Rivest-Shamir-Adleman (RSA), 65
  byteToString method, 189
  CCA, 191
  cryptosystem, 186, 189, 190
  decryption and encryption algorithm, 186
  factoring problem, 191
  homomorphic encryption, 190
  key generation algorithm, 186
  operations, 189
  polynomial time, 185
  RSA problem, 190
  standard data types, 189
  timing attacks, 191
RSA algorithm, 186, 189, 235, 238
RSA cryptosystem, 33, 141
  decrypation, 279
  encrypted message, 278
  keys generating, 285
  private and public keys, 285, 286, 288
RSA encryption system, 124, 190, 274–276
RSA Laboratories' PKCS#1 (PKCS #1 v2.2 (RFC 8017)), 131
RSA PRNG, 141
RSA problem, 124, 141, 190

S

Sample space, 119, 120
SEAL, 208
Sealed Classes (JEP 409), 32–34
Searchable encryption (SE), 205
  backward privacy, 212
  BuildIndex, 211
  cloud server, 210
  controlled searching, 211
  data owner, 210
  data user, 210
  encrypted data, 210

encrypted queries, 211
forward privacy, 212
innovations, 212
multi-keyword, 212
PKSE, 211, 212
query isolation, 211
retrieval component, 210
scheme and stores, 210
search, 211
security properties, 212
setup component, 210
SSE, 211, 212
trapdoor, 210, 211
SECG Test Suites, ECC Validation, 131
Secret key, 29, 45, 77, 90, 108–111, 166, 175, 176, 180, 183–185, 190, 198, 209, 221, 222, 233, 238, 239, 272, 273
Secure Hashing Algorithms (SHA) family, 155
Secure key distribution, 184
Secure online voting, 220
Secure quantum channel, 109
SecureRandom, 144, 147
Security-sensitive applications, 147
Service-to-service security, 45
SHA-2 family, 155
SHA-256 hash function, 159, 160
SHA-512 algorithm, 59, 60
Shared secret, 109, 114, 192, 193
Shift registers, 140
Short basis problem (SBP), 248
Shortest vector problem (SVP), 248, 249
Short independent vector problem (SIVP), 248
Shor's algorithm, 200, 208, 249
Signature schemes, 15–16, 131
digital signatures (*see* Digital signatures)
Six-State protocol, 114
Smart home security, 220
Somewhat homomorphic encryption (SWHE), 207
Sparse secret keys, 209
Standards for Efficient Cryptography Group (SECG), 131
Start-collision, 158
State compromise extension attacks, 142
Steganalysis, 119
Stop-and-go PRNGs, 141
Structured concurrency, 19, 21
Substitution approach, 7, 77
Substitution cipher, 11, 78, 86
Supply chain and Industrial IoT, 220
Symmetric cryptography, 77
Symmetric cryptosystems, 185
Symmetric encryption algorithms, 14
AES (*see* Advanced Encryption Standard (AES))
DES (*see* Data Encryption Standard (DES))
Symmetric encryption schemes, 183, 185
Symmetric searchable encryption (SSE), 211, 212

T

Test suites, 130–131
TFHE, 209
Theoretical cryptography
abstract models, 3
algorithm design, 2
computational hardness assumptions, 2
creation, novel cryptographic ideas and method, 3
cryptographic protocols, 2
cryptographic security principles, 3

Theoretical cryptography (*cont.*)
innovative research, 3
mathematical foundations, 2
security proofs, 2
Token-based authentication, 42
Transaction signing, 220
Transitioning to Post-Quantum Cryptography, 130
Transport Layer Security (TLS), 42, 185
Trapdoor, 210, 211
Triple Data Encryption Algorithm (TDEA/3DEA), 167, 168
Triple DES (3DES/TDES), 166, 167
True random number generators (TRNGs), 13, 135
Trusted authority (TA), 223
Twin field QKD (TF-QKD), 114
Two-factor authentication (2FA), 146
Two-step authentication, 222
.txt file, 270

U

Uniform Electronic Transactions Act (UETA), 243
Unkeyed cryptographic functions, 152–154
Unpadded RSA, 206, 207
User-friendly authentication modules, 49

V

Vector API (Second Incubator) (JEP414), 34
Vigenère cipher, 78
cryptanalysis, 89
18th century, 83
encryption table, 84
extend the key, 85
implementation, 86
OTP, 86
polyalphabetic substitution cipher, 84
technical aspects, 86
Vigenère encryption result, 89
Virtual threads (JEP 444), 21, 26, 35
Vote integrity and secrecy, 220
Voter authentication, 220

W, X, Y

Weak authentication, 219, 221, 222, 231
Weierstrauss form, elliptic curves, 128
Whirlpool hash function, 155

Z

Zero-knowledge proofs, 220, 222, 223, 231
Zettabytes era, 1